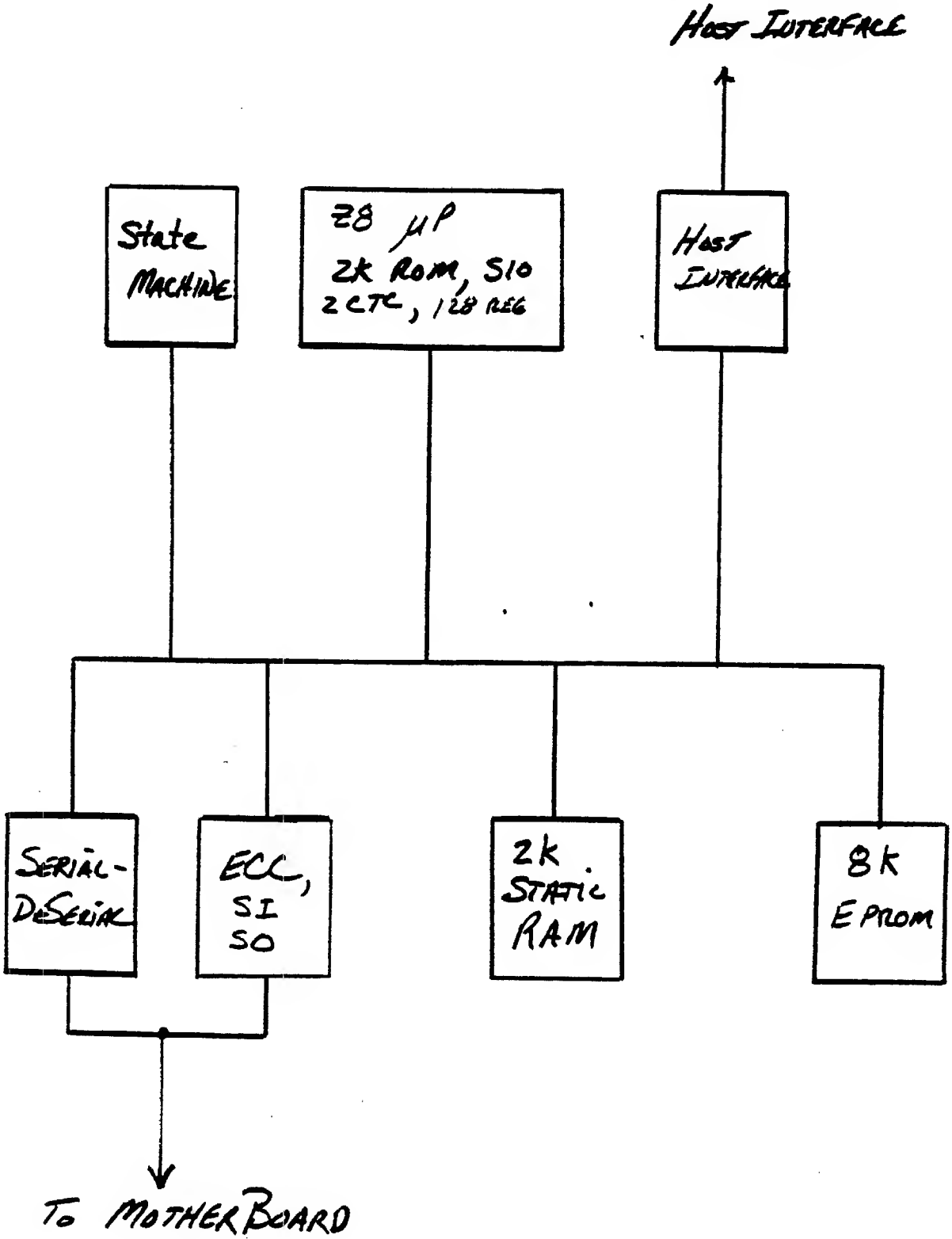
Widget Controller Block Diagram



28

1. INTELLIGENT CONTROLLER
   a) μ COMPUTER : RAM, ROM, SIO, CTC
   b) 4 MHz {7.3 ÷ 2}

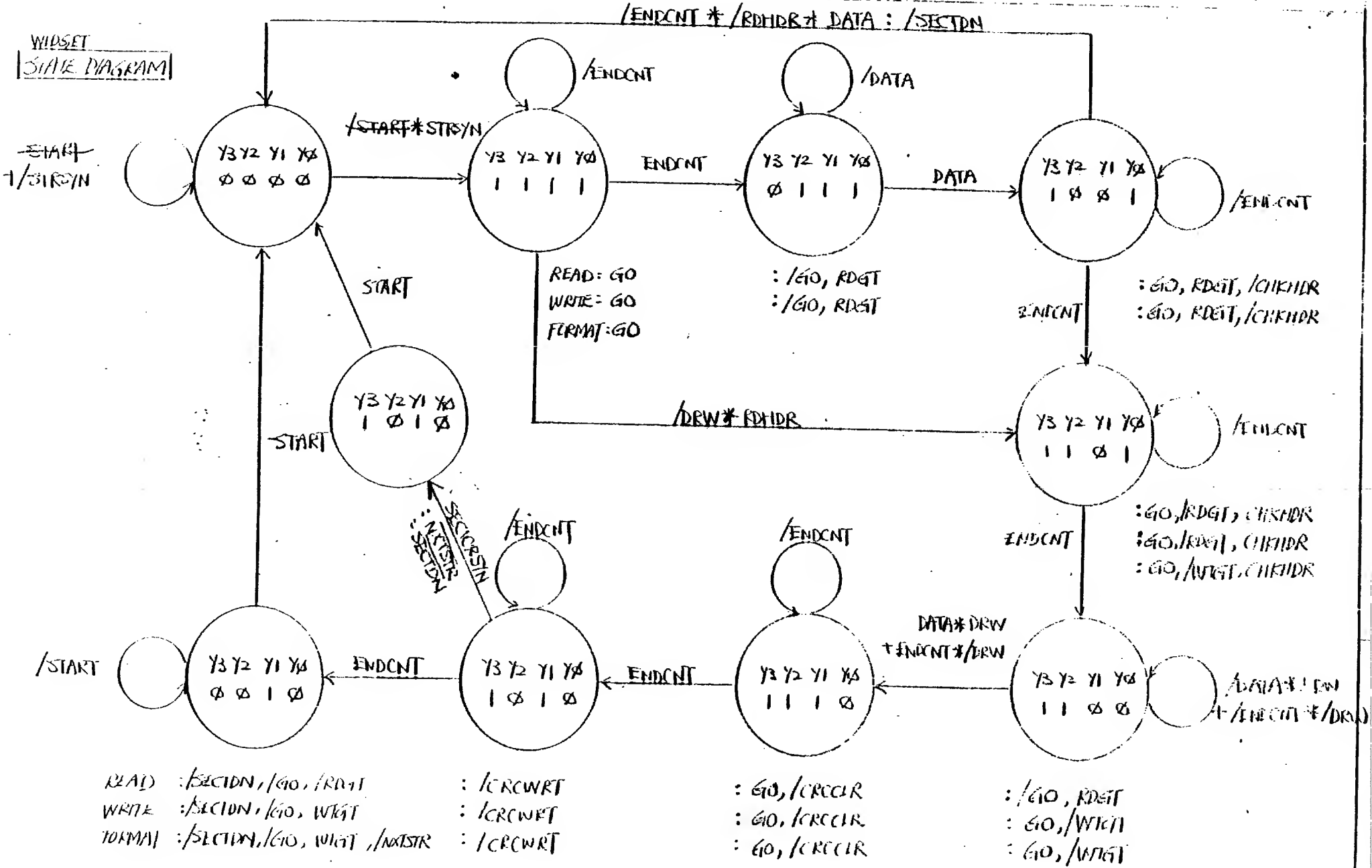
2. RECOVERY
   a) DEFECTS → SPARING
   b) NOISE
   c) SERVO ERRORS
   d) DATA CORRECTION

STATE MACHINE

1. SYNCHRONIZATION TO DISK
2. PERFORMS READ, WRITE, FORMAT, READ HEADER
3. CRC/ECC GENERATION
   a) ERROR DETECTION
4. LOADS/STORES WRITE/READ DATA TO/FROM DISK
5. POWER OK
   a) DETECTS WHEN +5V IS ~~WITH~~ WITHIN RANGE

WIDSET

STATE DIAGRAM



28 OPERATION : READ / READ (NO HEADER)

```
BEGIN
    MSEL1 := FALSE; MSEL0 := TRUE  { MEM <--> 28 }
    LOAD BUFFER WITH HEADER

          <#08> := HI-TRACK BYTE
          <#0B> := LO-TRACK BYTE
          <#0D> :=
                   <HI-NIBBLE> := HEAD SELECT
                   <LO-NIBBLE> := SECTOR NUMBER
          <#0E> := INVERT(<#08>)
          <#0F> := INVERT(<#0C>)
          <#10> := INVERT(<#0D>)
          <#11> := #00

    SET-UP STATE MACHINE
          MSEL1 := TRUE; MSEL0 := FALSE  { MEM <--> DISK }
          DM --> OUTPUT PORT := 0
          DRWL := FALSE  { DISK READ } ; FMENL := FALSE {NO FORMAT}
          IF NORMAL READ OPERATION
             THEN RDHDRH := FALSE
             ELSE RDHDRH := TRUE  { DON'T CARE ABOUT HEADER }
          POLL FOR SECTOR MARK  { PORT 3, BIT 2 }
          POLL FOR NOT(SECTOR MARK)
          STARTL := TRUE  { TURN STATE MACHINE ON }

    WAIT FOR SECTOR DONE OR TIMEOUT
          IF TIMEOUT THEN EXCEPTION
          IF SECTOR DONE
             THEN
                READ STATE MACHIN STATUS
                   IF STATE 0 THEN HEADER MISMATCH/GAP NOT ZERO
                   IF STATE 2
                      THEN
                         DISK DATA AT RAM ADR (#19 - #22C)
                         CRC AT RAM ADR(#22D - #22E)
                         ECC AT RAM ADR(#22F - #234)
                         IF CRC ERROR THEN EXCEPTION
                      ELSE
                         UNKNOWN STATE EXCEPTION
          STARTL := FALSE  { RESET STATE MACHINE
END
```

NOTE: IF THIS WAS A READ HEADER OPERATION THEN THE BYTES IN RAM ADDR <#0E> := #10 WERE REPLACED BY THE BYTES IN THE HEADER SPACE ON THE DISK. ~~THE HEADER READ LIKE IS~~

FIRMWARE

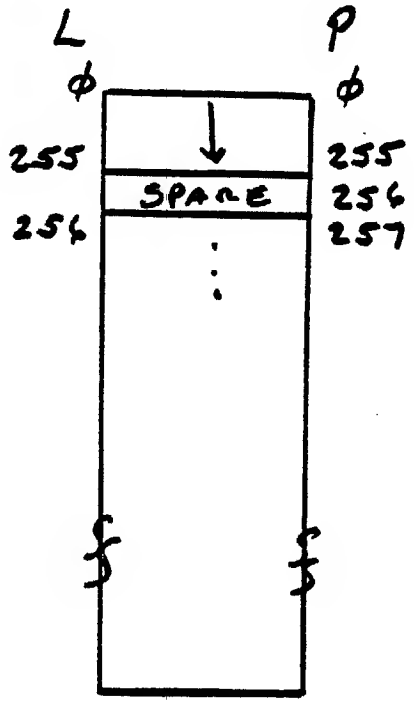
1. HOST INTERFACE PROTOCOL
   a) PROFILE, DIAGNOSTIC, MULTIBLOCK
2. CONTROLS STATE MACHINE, SERVO
   a) BASIC DISK FUNCTIONS
   b) POSITIONING
3. RECOVERY !!
4. PERFORMANCE

INITIALIZATION

1. BOOT STRAP A FEW 28 REGISTERS
2. TEST ALL 28 REGISTERS
3. STACK, CALL, RETURN TEST
4. INITIALIZE I/O; GLOBAL VARS
5. RAM TEST
6. EPROM TEST
7. MOTOR SPEED TEST {RELEASE BRAKE}
8. SECTOR COUNT
9. SERVO TEST
10. READ/WRITE TEST
11. FIND SPARE TABLE
12. SCAN

SPARING

Disk Blocks



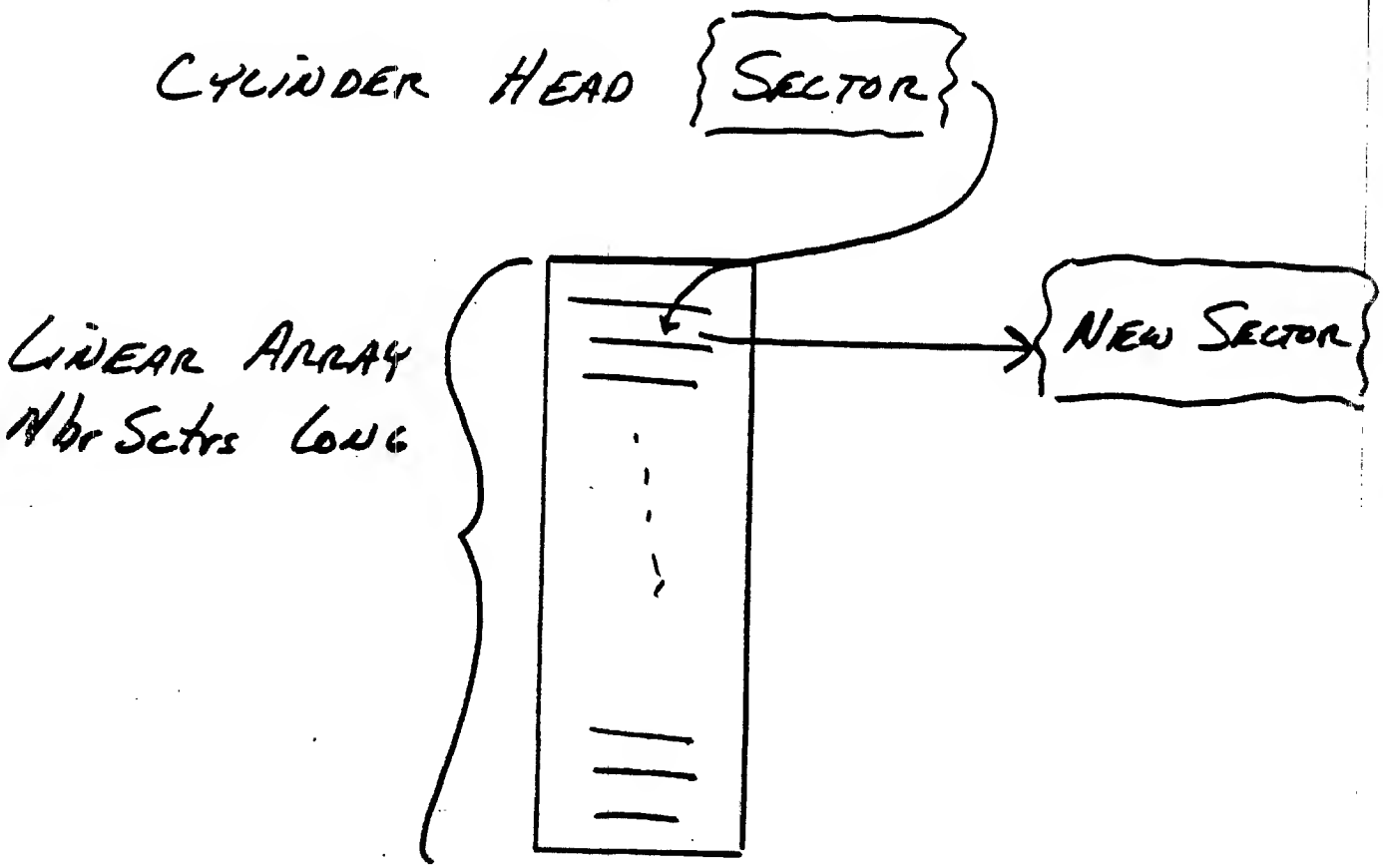
1. 10 MB → 1 SPARE / 256 Blocks
   20 MB → 1 SPARE / 512 Blocks
   40 MB → 1 SPARE / 1024 Blocks

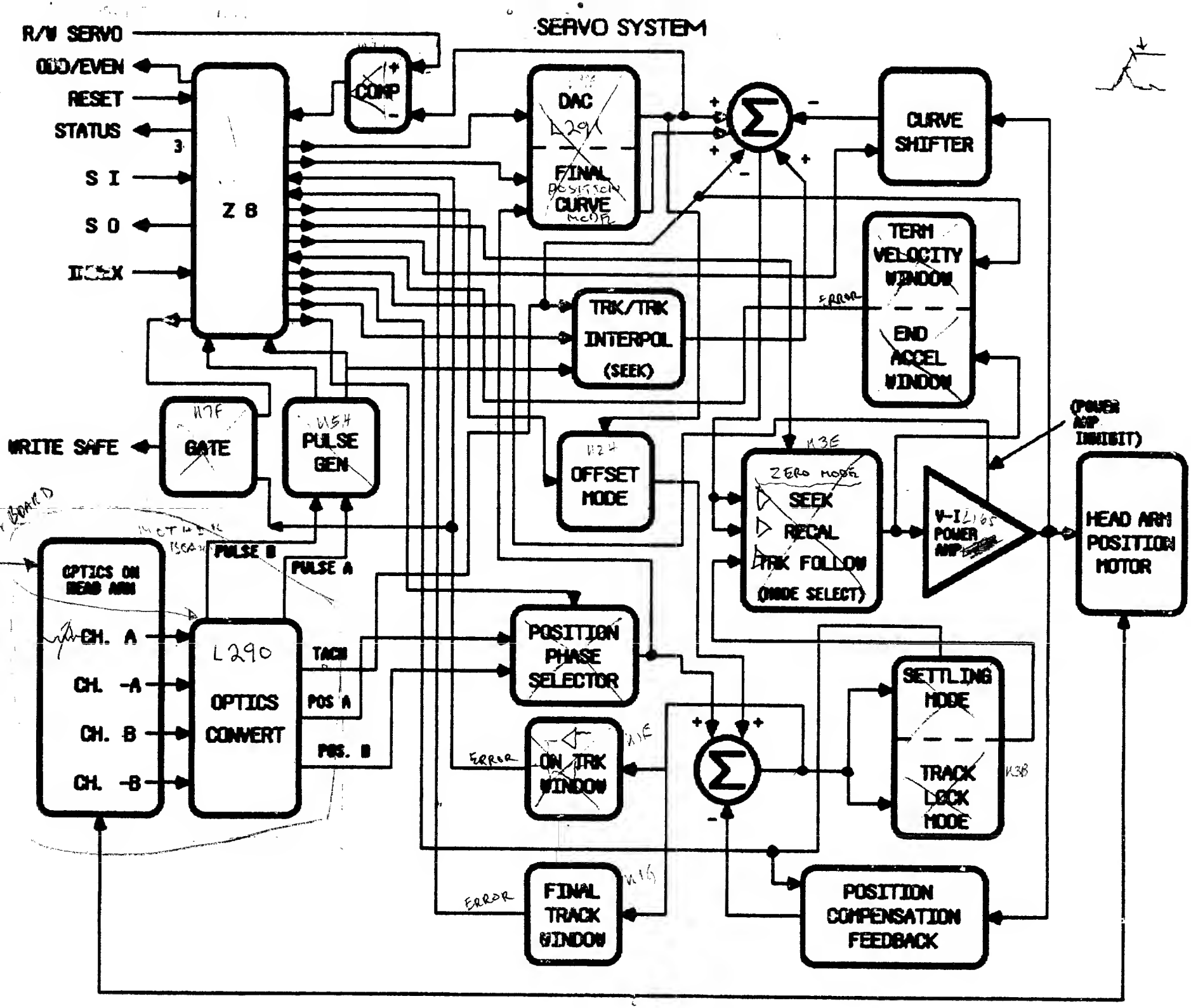
2. A Block is SPARED iff:
   a) VALID DATA is AVAILABLE
   b) THE Block is A <u>HARD</u> DEFECT

3. 76 TOTAL Blocks AVAILABLE FOR SPARING
   a) SPARE TABLE is LOCATED ON 2
   b) 74 LEFT FOR USER DATA

INTERLEAVING

1. ALL WIDGETS FORMATTED 2:1
2. CAPABICITY EXIST TO LOGICALLY INTERLEAVE 1:1 → Nbr Sctrs : 1
3. Offset Sector ϕ
   a) UP TO 16 SECTORS
   b) HEAD ϕ, HEAD 1 INDEPENDENT




SERVO SYSTEM
R/W SERVO
ODD/EVEN
RESET
STATUS
3
S I
S O
INDEX
Z 8
COMP
DAC
FINAL
CURVE
CURVE SHIFTER
TERM VELOCITY WINDOW
END ACCEL WINDOW
ERROR
TRK/TRK INTERPOL (SEEK)
WRITE SAFE
GATE
PULSE GEN
OFFSET MODE
ZERO MODE
SEEK
RECAL
TRK FOLLOW
(MODE SELECT)
V-I LOOP POWER AMP
(POWER AMP INHIBIT)
HEAD ARM POSITION MOTOR
OPTICS ON HEAD ARM
CH. A
CH. -A
CH. B
CH. -B
L290
OPTICS CONVERT
PULSE B
PULSE A
TACH
POS A
POS. B
POSITION PHASE SELECTOR
ON TRK WINDOW
ERROR
SETTLING MODE
TRACK LOCK MODE
FINAL TRACK WINDOW
ERROR
POSITION COMPENSATION FEEDBACK

Widget Firmware Specification
and
Theory of Operation

Revision 0.0-0

June 8, 1983

e#Profile#Interface:

A more complete description of the Apple/Profile interface may be found in document "EXTERNAL REFERENCE SPECIFICATION (E.R.S) PIPPIN HARDWARE" by Woolley and Wolfgang Dirks, dated April 16, 1981.

e are 5 control lines to/from the Apple Interface Card:

1. Parity
   This line is 1 bit of odd parity (even parity across the cable). The Interface Card is responsible for monitoring this signal: the controller calculates parity only when it sends a word across the bus; the controller does not check parity when a word is sent from the host, instead the parity bit is is generated once more on the controller side of the bus and then routed back to the host.

2. CMD (Command/Attention: Asserted by Host, Active high)
   This signal is one of two handshake signals across the interface bus. Keep in mind that even though the host and controller are two utonomous machines, the host is always considered the master and the controller the slave (in this configuration). When the host wishes to initiate a transfer to the controller it must first check if BSY (discussed below) is active. If BSY is active then the Host must wait (hopefully it will set a DeadMan timer and catch a "sick" controller) until BSY is no longer active.

3. BSY (Busy: Asserted by Controller, Active High)
   This signal is the dual of CMD, in other words this is the signal with which the controller can hold off the host for an indefinate period of time while it is "BUSY" perfotming some task.

4. STRB (Strobe: Asserted by the Host, Active High)
   Strobe is used to signal to the controller/host pair that data is valid on the bus.

5. R/W (Read/Write: asserted by the Host, Write is Active Low)
   This signal is used by the Host to indicate to the controller which direction data is to be going during a transmission. Read is used to direct data out of the controller into the host and the opposite condition is true for Write.

e are two modes of data transmission on the interface bus, single-byte and Direct Memory Access: multiple byte transfer, the number of bytes sf ed is up to the host). Both modes are invisible to the controller: e byte transmission is used to communicate directly with the controller (, read status or down load commands), while DMA is used to transfer data s to/from the controller's buffer space. In either case the controller NLY when it sees CMD become active and holds BSY active until it has eted it's task.

Profile#Communication#Protocol:

The following is an explanation of the protocol that is used to provide communication between the host and the controller:

Some explanation of the symbols that I am using is probably called for at this point.)

' ' : The bracket symbols mean that the information inclosed within them re manditory.

' ' : The square bracket symbols mean that the information inclosed is tional.

' : The vertical bar symbols is used to indicate an alternative or "OR" ondition. For example, A|B can be thought of as "Either A OR B".

'::=' : This symbols is used to indicate a definition or equivalence.

' ' : Curly brackets are used to denote comments.

' : The plus sign is as an addition symbol.

'NULL' : This key word indicates the empty set, or in some cases, the fact hat the function whose value is NULL can he ignored. An example is:

Argle-Bargle ::=<NULL>

ssentially you can forget that Argle#Bargle exists for this context.

THEN Instruction"Byte ::= <

Read"ID |
Read"Controller"Status |
Read"Servo"Status |
Send"Servo"Command |
Send"Seek |
Send"Restore |
Set"Recovery |
Soft"Reset |
Send"Park |
Diag"Read |
Diag"ReadHeader |
Diag"Write |
Set"Buffer"Ptr |
Read"SpareTable |
Write"SpareTable |
Format"Track |
Initialize"SpareTable |
Read"Abort"Stat |
Reset"Servo |
Scan >

ruction"Parameter"String ::= { This string is instruction dependent, and be formally specified at the same time as the individual instructions. }

kByte ::= { This byte is the ones-complement of the sum, in MOD-256 hmetic, of all the bytes including the Command"Byte }.

#ID ::= < S00 >

ruction#Parameter#String ::=<NULL>
diagnostic command requires Widget to deliver to the host some device
ific information. The structural layout of the data returned is:

CTURE Identity#Block

this identity block is defined by the data structures contained within
you will note, however, that a comment is given explaining the type of
ture for a given element and range of bytes (if the entire structure is
ght of as a linear array of bytes) that include the structure. An example
meString (first element to be defined below) which is a 13-character
i string, and is located in bytes S0 thru SC of the returned block.

NameString ::= <"Widget-10 " (13 Bytes/S00:S0C; Ascii String)>

DeviceType ::= <Device.Widget+Widget.Size+Widget.Type (3 Bytes/S0D:S0F)>
Device.Widget ::= <S0001 (2 Bytes/S10:S11)>
Widget.Size ::= <Size#10 | Size#20 | Size#40 (1 Nibble, Byte S12/bits 7:4)>
Size#10 ::= <S00>
Size#20 ::= <S10>
Size#40 ::= <S20>
Widget.Type ::= <System | Diagnostic (1 Nibble, Byte S12/bits 3:0)>
System ::= <S00>; This refers to the type of firmware that is imbedded in
Widget.
System firmware will not allow the host to Format, Write#SpareTable, or
Initialize#SpareTable; Diagnostic firmware will.

Diagnostic ::= <S01>

Firmware#Revision ::= <(2 Bytes/S10:S11)>

Capacity ::= <Cap#10 | Cap#20 | Cap#40 (3 Bytes/S13:S14)>
Cap#10 ::= <S004C00>
Cap#20 ::= <S009800>
Cap#40 ::= <S013000>

Bytes#Per#Block ::= < S0214 (2 Bytes/S15:16)>

Number#Of#Cylinders ::= < Cyl#10 | Cyl#20 | Cyl#40 (2 Bytes/S17:S18)>
Cyl#10 ::= <S0202>
Cyl#20 ::= <S0202>
Cyl#40 ::= <S0404>

Number#Of#Heads ::= <S02 (1 Byte/S19)>

Number#Of#Sectors ::= < Sctr#10 | Sctr#20 | Sctr#40 (1 Byte/S1A)>
Sctr#10 ::= <S13>

Read"Controller"Status ::= <SØ1>

Every time an operation completes { either successfully or exceptionally } Widget will return what I refer to as Standard"Status, thus allowing the Host system an opportunity to change it's flow of execution based on state of the Status. Normally, this Standard"Status is all that is necessary to ensure continuous operation. In the exceptional case, or when the Host system is emulating the controller's functions, additional information concerning the state of Widget is mandatory: without it the Host simply could not make an optimum choice in deciding a course of action.

Controller"Status is then a means for the Host system to interrogate Widget further. Each Status { with the exception of Abort"Status, which is a seperate command and is discussed later in this document } belongs to a homogeneous data structure: namely a four byte quantity containing a bit map representing the various exceptional conditions { active high } that is available as the first four bytes read from the controller upon completion of the current command.

There are seven status' available to the Host system. The Host requests a specific status by setting Instruction"Parameter"String to the value corresponding to the status needed.

```
IF ( Instruction"Byte = Read"Controller"Status )
        THEN Instruction"Parameter"String ::= <
                        Standard"Status |
                        Last"Logical"Block |
                        Current"Seek"Address |
                        Current"Cylinder |
                        Internal"Status |
                        State"Registers |
                        Exception"Registers >
```

The four byte response to each of the above status requests is of the form:

Result ::= < Byte0 Byte1 Byte2 Byte3 >

Last#Logical#Block ::= < S01 >

Byte0 ::= < S00 >

Byte1 ::= < { Most Significant Byte of Logical#Block#Number } >

Byte2 ::= < { Middle Byte of Logical#Block#Number } >

Byte3 ::= < { Least Significant Byte of Logical#Block#Number } >

Current#Seek#Address ::= < S02 >

Byte0 ::= < Most Significant Cylinder Address >

Byte1 ::= < Least Significant Cylinder Address >

Byte2 ::= < Head Address >

Byte3 ::= < Sector Address >

Current#Cylinder ::= < S03 >

{ The Current#Cylinder differs from the Current#Seek#Address in that it is perfectly reasonable for the Servo to have placed the heads on another track under certain circumstances; for example, the drive may have been bumped }

Byte0 ::= < Most Significant Cylinder address >

Byte1 ::= < Least Significant Cylinder address >

Byte2 ::= < S00 >

Byte 3 ::= < S00 >

Bit1: Ram#Space#Ø enabled
Bit0: IF active THEN controller LED should be ON

{ The Ram#Spaces mentioned above are 5 2k address spaces overlayed on top of one another to provide the controller with the ability to keep several disk blocks temporarily resident in ram. At the time of this writing, however, only Ram#Space#Ø is being used. }

Byte3 ::= < { Register: Controller#Status#Port }
Bit7: CrcError { active low }
{ this bit is valid ONLY when the controller state machine is NOT in reset, which should be every time that this bit is read by the host. Therefore, if this status bit indicates a CrcError, then something has croaked. The normal way for the host to check if a Crc or Ecc error has occured is to examine Status: Exception#Registers which are dicussed below. }

Bit6: Write#Not#Valid { active low }
{ as in CrcError, this bit is valid only when the state machine is NOT in reset. The information expressed by this bit is converted into a type of ServoError, which is found in Status: Exception#Registers. }

Bit5: ServoReady
Bit4: ServoError
{ the servo status bits listed above are further explained in Appendix A: Servo Processor Documentation. Essentially the two bits combine to form four possible servo states; the normal condition is ServoReady AND (NOT ServoError). }
Bit3:Ø Current controller state-machine state.
{ as in CrcError and Write#Not#Valid, these status bits are valid only when the state machine is NOT in reset, and should read ØØ any other time. }

On the surface it appears that this byte is of limited use for non real- time situations. It is, however, invaluable in trying to decide if the Servo Processor is healthy, wealthy, and wise. It also provides a means for diagnosing a sick state machine.

Read#Servo#Status ::= < SØ2 >

Instruction#Parameter#String ::= < Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 > < $ØØ..$Ø8 >

This status command is used to interrogate the Servo Processor in much the same way that Read#Controller#Status is used. In fact, the form of the result is the same four byte bit-mapped quantity.

This command is of particular value to a diagnostician that is interested in 'picking-about' with the servo processor without dismantling Widget as a subsystem. Refer to Appendix A: Servo Processor Documentation for a complete description of the various status' available and their resulting bit descriptions.

Send#Servo#Command ::= < SØ3 >

Instruction#Parameter#String ::= < Byte0 Byte1 Byte2 Byte3 >

Normally, the Host will allow the controller to manipulate the servo processor in order to perform useful (or maybe not so useful!) work. For example, let's suppose that the Host system wishes to move the disk drive heads from one track to another. Under normal operating conditions the preferred way to perform this task is to use the Send#Seek command (explained below). However, the Host has the capability to bypass the controller and direct the servo processor. Indeed, the Host can issue the servo command to position the heads (via the Send#Servo#Command) so that the seek is completly transparent to the controller. The implication of this command is that the Host can gain even more control of the system if it so chooses.

A more complete description of the Servo Commands can be read in Appendix A: Servo Processor Documentation.

Byte0 ::= < S#Command + S#Direction + Hi#Magnitude >

S#Command ::= <

Offset
Diagnostic
DataRecal
FormatRecal
Access
Access#Offset
Home

Offset ::= < S1Ø >

The Offset command allows the Host to microstep the heads in either a positive or negative direction from the center of the track. The Widget Firmware does not make use of this feature! I have instead left this to a more specific data recovery program that is run by the Host. The value and direction of the microstep are sent to the Servo Processor in Byte2.

S#Direction ::= < Positive | Negative >

Positive ::= < $Ø4 { move the heads toward the outside diameter } >
Negative ::= < $ØØ { move the heads toward the inside diameter } >

Hi#Magnitude ::= < Ø | 1 | 2 | 3 { move the heads a multiple of 256 tracks } >

Byte1 ::= < Low#Magnitude ::= Ø..255 >
Hi#Maginitude + Low#Maginitude, and S#Direction establish the relative distance the heads must move to arrive at the target track.

Byte2 ::= < Offset#Direction + Auto#Offset#Switch + Offset#Magnitude >

This command byte, when used with the Offset command, establishes the degree and direction of microstepping.

Offset#Direction ::= < Positive | Negative >
Positive ::= < $8Ø { offset towards outside diameter } >
Negative ::= < $ØØ { offset towards inside diameter } >

Auto#Offset#Switch ::= < ON | OFF >
ON ::= < $4Ø { turn automatic track centering on without an access command } > OFF ::= < $ØØ { do not auto track center on this command } >

Offset#Maginitude ::= < Ø..32 >

Byte3 ::= < Baud#Rate + Power#On#Reset >

Baud#Rate ::= < 19.5k#Baud | 57.6k#Baud >

The servo 'comes up' at 19.5k baud because of the test equipment used on it before it is integrated into a system. Once it is running with a controller, however, it is run continuosly at 57.6k baud. This parameter is also a bit misleading in that once the servo has been told to go to 57.6k it will forever more ignore this parameter: in other words it is impossible to go from the higher baud rate to the lower without reseting the servo processor.

Send#Seek ::= < $04 >

Instruction#Parameter#String ::= < HiCyl LoCyl Head Sector >

Widget's Send#Seek command allows the Host system to place the heads over any track on the disk. The value of the seek address sent in the parameter string is used read/write a block of data using the diagnostic commands for those functions. For example, for the Host to read Cylinder 1, Head Ø, Sector 18 a Seek#Command would be issued for that combination of cylinder, head, and sector ($0001 ØØ 12) followed by a Diag#Read explained below).

Set#Recovery ::= < $Ø6 >

Instruction#Parameter#String ::= < ON | OFF >

ON ::= < $Ø1 >
OFF ::= < $ØØ >

To the best of my ability I have attempted to make the exception handling characteristics of Widget a binary set: either Widget handles everything, or the Host system does. The command Set#Recovery is the Host's link with this all or nothing world in that it is through this instruction that the Host can gain control of the media. When Widget comes up after being reset it assumes control and sets Recovery to be ON. The Host system must overtly change this state (via Set#Recovery) if it wishes to emulate a different exception handling criteria. Once Recovery is OFF, the controller will always fail in an operation if an exception occurs: the Host system MUST assume responsibility for ALL error handling.

Send#Park ::= < $08 >

Instruction#Parameter#String ::= < NULL >

When the Host issues a Send#Park command to the controller the results are that that the heads are moved off the data surface and held very near the inside diameter crash stop. The difference between this command and the Send#Servo#Command: Home is that Home is performed 'open-loop' with the crach stop as it's reference point, while Send#Park is an access command to a specific track. The net result is a fairly hefty saving of time: the access command can be an order of magnitude quicker than Home/Recal.

Sync ::= < $0100 >

Set#Buffer#Ptr ::= < $0C >

Instruction#Parameter#String ::= (< HiAdr > < LowAdr >)

HiAdr ::= < Most significant byte of buffer address >
LowAdr ::= < Least significant byte of buffer address >

The Set#Buffer#Ptr command is externally (in the Host's point of view) identical to a Read command: The Host/Controller handshake CMD/BSY a few times with the appropriate responses and the Host reads from the controller's buffer area to receive data. In this instruction sequence, however, the host does not read a block of data from the disk, but rather an arbitrary number of bytes from an arbitrary location in the controller's ram space. The Host also has the ability to write to this ram space - in effect trashing all of the controller's brains if it so desires. The intent of this command is to allow the Host to perform diagnostics or read variables that are otherwise not available.

though of as a linear array of bytes, the a Ptr is used to index into that array). To arrive at the actual index value within the Heap, the Ptr must first be multiplied by four.

When a disk is formatted and fresh data is being written to it, each logical block is asigned the first available physical block on the disk. Therefore you would expect that LogicalBlock(Ø) would occupy PhysicalBlock(Ø), L(1) --> P(1), etc. There are instances, however, when a block of data must be relocated to another space on the disk that does not follow the original progression (for example, the original space was defective). In order to 'find' these relocated blocks in the future a record must be kept as to where all these relocated blocks have been put. This record takes the form of 128 linked lists having the form HeadPtr(n) --> LinkedList(n), where n := Ø..127. The algorithm for deciding whether or not a LogicalBlock has been relocated is to extract bits 16:10 from the LogicalBlockNumber and use it as an index into the HeadPtr#Array. If the HeadPtr associated with this index value is Nil then LogicalBlock has not been relocated else use HeadPtr.Ptr to search the linked list corresponding to this HeadPtr value. Now to decide if the LogicalBlock has been relocated a test must be made as the linked list is traversed by comparing the LogicalBlockNumber's bits 9:Ø to the current list element's token value. If they match then LogicalBlock has been relocated and it's new position is a multiple of the list element's position in the Heap.

```
SpareCount ::= < SØØ..S4C >
BadBlockCount ::= < SØØ..S4C >

BitMap ::= < ARRAY[ Ø..S4B ] of Bits >
     The bit map is used to keep a record of which spare blocks are
     occupied, and their locations on the disk.

Heap ::= < ARRAY[ Ø..S4B ] of ListElement >

     ListElement ::= (
               < Nil+Used+Useable+Spr#Type+Data#Type >
               < Token >
               < Ptr > )

          Nil ::= < S8Ø { IF Nil THEN End#Of#Chain } >
          Used ::= < S4Ø >
          Useable ::= < S2Ø >
          SprType ::= < Spare | BadBlock >
               Spare ::= < S1Ø >
               BadBlock ::= < SØØ >
          Data#Type ::= < Data | SpareTable >
               Data ::= < S02 >
               SpareTable ::= < S08 >
```

Write#Spare#Table ::= < S0E >

Instruction#Parameter#String ::= (< SF0 > < S78 > < S3C > < S1E >)

This command allows the Host to 'force' a new spare table on the controller, and is executed just like any of the other write commands (the data in this case MUST conform to the structure presented in Read#SpareTable). The data sent to the controller is written to the two spare table locations on the disk.

Initialize#SpareTable ::= < $10 >

Instruction#Parameter#String ::= (
< Format#Offset >
< Format#InterLeave >
< PassWord >

Format#Offset ::= < $00..Number#Of#Sectors >

Format#InterLeave ::= < $00..$06 (interleave factor) >

PassWord ::= (< $F0 > < $78 > < $3C > < $1E >)
This command form the Host instructs the controller to 'wipe the slate clean' as far as the SpareTable is concerned. The initialized table is written to disk.

Reset#Servo ::= < $12 >

Instruction#Parameter#String ::= < NULL >

Reset#Servo allows the host to initialize the servo processor without having to power the device down. The controller will automatically reset the Servo, check for valid initial conditions and perform a Data#Recal.

SCAN := < $13 >

INSTRUCTION_PARAMETER_STRING ::= <NULL>

THIS COMMAND READS EVERY LOGICAL BLOCK ON THE DISK. IF A BAD BLOCK IS FOUND IT IS ADDED TO THE SPARE TABLE AS EITHER A SPARE (IF VALID DATA CAN BE DERIVED FROM THE BLOCK) OR AS A BAD BLOCK. REFER TO THE SECTION ON EXCEPTION HANDLING FOR MORE INFORMATION ON SPARING.

Sys#Read ::= < S00 >

Instruction#Parameter#String ::= (< Block#Count > < LogicalBlock >)

Block#Count ::= < S00..S08 >
This parameter is the number of blocks to be read that follow sequentially from LogicalBlock. It is assumed that one block (LogicalBlock) will be read, making the Block#Count the number of blocks following the first one that is to be read, also.

LogicalBlock ::= < L#10MB | L#20MB | L#40MB >
L#10MB ::= < S000000..S004BFF >
L#20MB ::= < S000000..S0097FF >
L#40MB ::= < S000000..S012FFF >

HANDSHAKE PROTOCOL

Both Widget and ProFile share the same Host interface scheme, and therefore a lot in common when it comes to trying to communicate with the Host system. ProFile's protocol is documented in 'ProFile Communication Protocol' for those of who wish to read it.

The actual sequence of events can be portayed as follows:

```
Protocol*Sequence ::= (
                < Initial*HandShake >
                < Command*DownLoad >
                < Response*HandShake >
                [ Data*Received*HandShake ]
                < Final*HandShake > )
```

Initial*HandShake ::=

1. Host asserts CMD, sets data direction to read
2. Controller asynchronously responds by:
   a. Writing $01 to the Host
   b. Asserting BSY

3. If the Host recognizes the controller response, it will respond by:
   a. Writing a $55 to the controller
   b. Otherwise it will write a $AA
   c. In either case the Host will de-assert CMD.

4. The controller will respond to the Host by:
   a. In either case (whether the Host responded with a $55 or $AA or anything else) the controller will eventually end up waiting for the next instance of CMD.
   b. If the response was a $55 then the controller will be a 'captive' audience, anxiously awaiting instructions from the Host as to what to do next.
   c. Otherwise, the controller will Abort, and leave Standard Status saying so in it's buffer where the host can read it. The state of the command sequence for the controller then becomes Initial*HandShake, and the Host should read it's best to

5. The controller then asserts BSY

6. Assuming the Host accepts the response from the controller, it will respond by writing $55 back to the controller and then de-asserting C/D.

7. The controller will then continue executing the command.

Final HandShake ::=

1. When the controller finishes with the execution of the instruction,
   it will put the latest Standard Status in a location in it's buffer
   where it will be accessible to the Host (as well as any data that
   might be a result of the command execution).

2. The controller then de-asserts BSY

COMMAND SUMMARY

ProFile#Commands:

```
ProFile#Read ::= ( <S00> < 3 bytes LogicalBlock > )
ProFile#Write ::= ( <S01> < 3 bytes LogicalBlock > )
ProFile#WrVerify ::= ( <S02> < 3 bytes LogicalBlock > )
```

Diagnostic#Commands:

```
Read#ID ::= ( <S12> <S00> <SED> )
Read#Controller#Status ::= ( <S13> <S01> < Status > < CheckByte > )
Read#Servo#Status ::= ( <S13> <S02> < Status > < CheckByte > )
Send#Servo#Command ::= ( <S16> <S03> < 4 command bytes > < CheckByte > )
Send#Seek ::= ( <S16> <S04> < 4 bytes cyl/head/sector > < CheckByte > )
Send#Restore ::= ( <S13> <S05> < Data/Format Recal > < CheckByte > )
Set#Recovery ::= ( <S13> <S06> < On/Off > < CheckByte > )
Soft#Reset ::= ( <S12> <S07> <SE6> )
Send#Park ::= ( <S12> <S08> <SE5> )
Diag#Read ::= ( <S12> <S09> <SE4> )
Diag#ReadHeader ::= ( <S13> <S0A> < Sector > < CheckByte > )
Diag#Write ::= ( <S12> <S0B> <SE2> )
et#Buffer#Ptr ::= ( <S14> <S0C> < 2 bytes buffer address > < CheckByte > )
Read#SpareTable ::= ( <S12> <S0D> <SE0> ) < PASSWORD >
Write#SpareTable ::= ( <S12> <S0E> <SDF> ) < PASSWORD >
Format#Track ::= ( <S14> <S0F> < Offset > < InterLeave > < CheckByte > )
Initialize#SpareTable ::=
        ( <S14> <S10> < Offset > < InterLeave> < CheckByte > ) < PASSWORD >
Read#Abort#Stat ::= ( <S12> <S11> <SDC> )
Reset#Servo ::= ( <S12> <S12> <SDB> )
Scan ::= ( <S12> <S13> <SDA> )
```

System Commands:

```
Sys#Read ::=
        ( <S26> <S00> < BlkCnt > < 3 bytes LogicalBlock > < CheckByte > )

Sys#Write ::=
        ( <S26> <S01> < BlkCnt > < 3 bytes LogicalBlock > < CheckByte > )

Sys#WrVerify ::= ( <S25> <S02> < 3 bytes LogicalBlock > < CheckByte > )
```

< PASSWORD > ::= < $FD $F8 $3C $1E >

READ/WRITE EXCEPTIONS

There are occasions when the a spot on the disk surface becomes unuseable, or for some reason causes the data stored in that area to change. To handle this type of exception Widget is equiped with 2 error detecting devices and 1 error correcting device { although Ecc is both error detecting and error correcting }. Widget uses a sixteen-bit crc polynomial { CRC-16 } to detect all single-burst errors less than sixteen bits in length, almost all single-burst errors of sixteen bits, and most single-burst errors greater than sixteen bits in length. A 48-bit ecc polynomial is also used that has error detecting properties similar to that of the crc polynomial, except that it handles burst of up to 48 bits. It can also correct single-error bursts up to twelve bits in length.

When a block read, if the first read is successful { no errors } then the data is transfered to the Host, thus completing it's command. Suppose, however, that the block is not read successfully the first time. The causes of this exception are 4:

1. Servo Error: this exception is handled by leaving the read routine and getting in touch with the Servo Processor to see if things can be straightened out. Once the controller is convinced that the Servo is well and that the heads are positioned where thve should be, it retries the read.

2. The state machine indicates that it is in the wrong ending state. This is considered a catastrophic exception an the controller will abort.

3. The state machine indicates that a matching header was not found. Before making this decision the state machine searches the track twice for a match header. To handle this exception the controller reads a header from the track that the heads are currently positioned over and tries to determine if the heads are positioned correctly. If they are, then it is assumed that target block's header is faulty and the track will be spared. If no header can be read from the track it can be determined if the heads are positioned correctly or if all headers on the track are shot. In this case the controller will issue a data recal and seek back to the target location and retry. If a header still can not be found the block will be spared.

4. The state machine indicates that a crc or ecc error has occured. The controller will automatically retry 9 times { a total of 10 reads }. If a successful read is encountered during this retry session the controller will save the valid data. At the end of all the retries, if the number of bad reads was 2 or less then the block is transfered to the Host. If the number is between 2 and 10 then the data is still returned to the Host, but the controller goes back to the target block and performs a WriteVerify with the valid data; if the block fails the verify then it is spared. If the number of bad reads is 10 then the ecc correction algorithm is applied to the result of the last retry. If the data is

MISCELLANEOUS

Parking:

To guard against any mishaps when power is shut off to Widget, there is a mechanism in the firmware that takes the heads off the data area of the disk after a period of idleness. This mechanism is known as 'parking'. Unfortunately, it is possible for parking to synchronize with periodic uses of the drive by the Host, causing a mild form of thrashing brought about by the constant seeking needed to move the heads between the park position and the target position. It was determined empirically on ProFile that a good compromise delay time to park is 3 seconds and that time hold for Widget.

Arm Sweep:

To protect the head-arm bearings from too many short seeks (this causes a possible migration of lubrication away from the surfaces that are meant to be lubricated) the arm is swept the complete width of the disk data surface every 2048 seeks.

Self Test:

When the controller comes up from being reset it performs the following selftest functions:

Hard Fail

1. Register Test
   Write and verify one's and zero'z to all registers; halt if failure
2. Stack Test
   Check push/pop, call/return capabilities; halt if failure

Fails that Boot ROM can show?

3. Ram Test
   Write ones and zeros to all ram locations; don't allow ProFile or System commands if failure.
4. Eprom Test
   Check external eprom banks 0 and 1 for check byte; don't allow ProFile or System commands if failure.
5. Motor Speed
   Check time from index to index; don't allow ProFile or System commands if failure.
6. Track Count

   Seek to track 0 and read a header, if no heeader found then format recal and count tracks; don't allow ProFile or System commands if failure.
7. Spare Table

   Find both spare tables and write verify them; don't allow

WIDGET SERVO FUNCTIONAL OBJECTIVE

I. BASIC SERVO FUNCTIONS

Widget servo control functions are handled by a Z8 microprocessor. The Z8 handles all I/O operations, timing operations and communication with a host controller. Control functions to the Z8 Servo Controller are made through the serial I/O.

The following commands for the Widget servo are:

A. HOME - not detented, heads off data zones located at the inner stop.

B. RECAL - detented at one of two positions.

1. FORMAT RECAL: 32, -0, +3 tracks from HOME use only during data formatting.

2. RECAL: 72, -0, +3 tracks from HOME use to initialize home position after power on or following an access or any other error.

C. SEEK - coarse track positioning of data head to any desired track location.

D. TRACK FOLLOWING - heads are detented on a specific track location and the device is ready for another command.

E. OFFSET - controlled microstepping of fine position system during TRACK FOLLOWING (two modes).

1. COMMAND OFFSET - direction and amount of offset is specified to the servo.

2. AUTO OFFSET - command allows the servo to automatically move off track by the amount indicated by the embedded servo signal on the data surface (disk).

F. STATUS - command can read servo status.

G. DIAGNOSTIC - not implemented.

See Table 1 for the actual command description. With the present command structure a SEEK COMMAND can be augmented with an OFFSET COMMAND. Upon completion of a seek, the offset command bit is tested to determine if an offset will occur following a seek (either auto or command offset).

Part of the communication function requires a specific protocol between the servo Z8 processor and the external controller.

Servo control and communication are described in CHART I. This chart illustrates the basic sequencing and control operations. Chart I does not illustrate the servo error handling or command/protocol handling functions. Error handling is described in Section IV and illustrated by CHART II.

III. Z8 SERVO PROTOCOL

The protocol between the Z8 SERVO microcomputer and the CONTROLLER is based on five I/O lines. Two of the I/O lines are serial input (to Z8 servo from controller) serial output (from Z8 servo to controller). Data stream between the Z8 servo and controller is 8 bit ACSII with no parity bit (the fifth byte of the command string contains check sum byte use for error checking). There are three additional output lines between the Z8 servo used as control lines to the controller. Combining the two serial I/O lines and the three unidirectional port lines generates the bases of the protocol between the Z8 servo and controller. The important operations between the Z8 servo and controller are:

1. Send commands to Z8 servo.

2. Read Z8 servo status.

3. Check validity of all four command bytes.

4. I/O timing signals between the Z8 servo and controller.

5. Z8 servo reset.

Sequencing the Z8 servo controller is an important process following a Power Up (Power On Reset) or if the controller should issue a Z8 Servo Reset at any time. After a Z8 Servo Reset is inhibited the Z8 I/O ports and internal register are initialized. This takes approximately 75 msec after the Z8 Servo Reset is inhibited. The protocol baud rate is automatically set to 19.2KB and then the system is parked at HOME position and SIO READY is set active. ***IMPORTANT***. If the desired baud rate needs to be increased to 57.6KB; **after a Z8 Servo Reset is the ONLY time this can be done***. Once set to 57.6KB the communication rate remains at 57.6KB until a Z8 Servo Reset occurs. Setting 57.6KB is achieved as follows:

1. Z8 Servo "Power On or Controller" Reset

2. Wait for SIO Ready

3. Send a READ STATUS COMMAND as follows:

   BYTE 1 = $ 00
   BYTE 2 = $ 00
   BYTE 3 = $ 00
   BYTE 4 = $ 37

2. During Seek mode (velocity control only) access time-out. If a Seek function exceeds 150 msec then an access time-out occurs.

3. During Settling mode (following a Recal, Seek, or Offset) if there is excessive On Track pulses (3 crossings) indicating excessive head motion a Settling error check will occur.

4. During a command transmission if a communication error occurs (check sum error).

5. During a command tansmission if a invalid command is sent.

28 SERVO COMMAND BYTES
TABLE 1

BYTE 1: COMMAND BYTE (DIFCNTH)

```
          ---
         |B7
ommand   |B6
yte      |B5
         |B4
          ---
          ---
         |B3 -<- not used
ccess    |B2 -access direction
ts       |B1 -hi diff2 (512)
         |B0 -hi diff1 (256)
          ---
```

B7	B6	B5	B4	FUNCTIONS
1	0	0	0	access only
1	0	0	1	access with offset
0	1	0	0	normal recal to trk 72
0	1	1	1	format recal to trk 32
0	0	0	1	offset-trk following
1	1	0	0	home-send to IO stop
0	0	1	0	diagnostic command
0	0	0	0	read status command

access direction = 1 (FORWARD: toward the spindle)
= 0 (REVERSE: away from the spindle)

hi diff2 (512) = 1 (512 tracks to go)
= 0 (not set)

hi diff1 (256) = 1 (256 tracks to go)
= 0 (not set)

BYTE 2: DIFF BYTE (DIFCNTL)

command BYTE 2 contains the LOW ORDER DIFFERENCE COUNT for a seek

```
 ---
|B7 -bit7= 128 tracks
|B6 -bit6= 64  tracks
|B5 -bit5= 32  tracks
|B4 -bit4= 16  tracks
|B3 -bit3= 8   tracks
|B2 -bit2= 4   tracks
|B1 -bit1= 2   tracks
|B0 -bit0= 1   track
 ---
```

Z8 SERVO COMMAND BYTES
TABLE I

BYTE 5: CHECKSUM BYTE (CKSUM)

[B7 B6 B5 B4 B3 B2 B1 B0]

results of the transmitted CHECKSUM BYTE are derived as:

$$(\text{BYTE 1} + \text{BYTE 2} + \text{BYTE 3} + \text{BYTE 4}) = \text{CHECKSUM BYTE}$$

(+) is defined as the addition of each BYTE

$\overline{(\text{BYTE})}$ is defined as the compliment of the BYTE (1-d)

1. The SERVO STATUS lines (SIO RDY, SERVO RDY, SERVO ERROR) must have the following conditions in order to send the listed Z8 COMMANDS:

Z8 SERVO CMD	HEX	SIO RDY	SERVO RDY	SERVO ERROR	
access(only)	8X	1	1	0	
access(offset)	9X	1	1	0	
recal(data)	40	1	X	X	
ecal(format)	70	1	X	X	
ark	C0	1	X	X	
ffset(detent)	10	1	1	0	
tatus	00	1	X	X	
iagnostic	20				not implimented

X= either 0,1

Widget Firmware Specification
and
Theory of Operation

Revision 1.Ø-Ø
October 16,1983

Written by Rodger Mohme
Ms-2ØD x4879

Widget is Apple's in-house name for the latest in a line of Winchester hard disks. This current version is available with 10.1 MB of storage (formatted).

Widget has been designed as a complete, self-contained intelligent subsystem. The purpose of this document is to explain in detail how this subsystem behaves within the complete system environment.

Apple_Profile_Interface:

A more complete description of the Apple/Profile interface may be found in the document "EXTERNAL REFERENCE SPECIFICATION (E.R.S) PIPPIN HARDWARE" by Dick Woolley and Wolfgang Dirks, dated April 16, 1981.

There are 5 control lines to/from the Apple ProFile Interface Card:

1. Parity
   This line is 1 bit of odd parity (even parity across the cable). The Interface Card is responsible for monitoring this signal: the controller calculates parity only when it sends a word across the bus; the controller does not check parity when a word is sent from the host, instead the parity bit is is generated once more on the controller side of the bus and then routed back to the host.

2. CMD (Command/Attention: Asserted by Host, Active high)
   This signal is one of two handshake signals across the interface bus. Keep in mind that even though the host and controller are two autonomous machines, the host is always considered the master and the controller the slave (in this configuration). When the host wishes to initiate a transfer to the controller it must first check if BSY (discussed below) is active. If BSY is active then the Host must wait (hopefully it will set a DeadMan timer and catch a "sick" controller) until BSY is no longer active.

3. BSY (Busy: Asserted by Controller, Active High)
   This signal is the dual of CMD, in other words this is the signal with which the controller can hold off the host for an indefinate period of time while it is "BUSY" performing some task.

4. STRB (Strobe: Asserted by the Host, Active High)
   Strobe is used to signal to the controller/host pair that data is valid on the bus.

5. R/W (Read/Write: asserted by the Host, Write is Active Low)
   This signal is used by the Host to indicate to the controller which direction data is to be going during a transmission. Read is used to direct data out of the controller into the host and the opposite condition is true for Write.

Profile Communication Protocol:

The following is an explanation of the protocol that is used to provide communication between the host and the controller:

(Some explanation of the symbols that I am using is probably called for at this point.)

'<,>' : The bracket symbols mean that the information inclosed within them are manditory.

'[,]' : The square bracket symbols mean that the information inclosed is optional.

'|' : The vertical bar symbols is used to indicate an alternative or "OR" condition. For example, A|B can be thought of as "Either A OR B".

':=' : This symbols is used to indicate a definition or equivalence.

'{,}' : Curly brackets are used to denote comments.

'+' : The plus sign is as an addition symbol.

'NULL' : This key word indicates the empty set, or in some cases, the fact that the function whose value is NULL can be ignored. An example is:

Argle-Bargle ::= < NULL >

Essentially you can forget that Argle_Bargle exists for this context.

PROFILE_COMMANDS

These commands are currently by the SOS driver. Widget is designed to be backwards compatible with the current ProFile driver, and to that end there exists the three ProFile system commands: Read, Write, and Write_Verify.

Profile Commands:

Opcode	Definition
$ØØ	Read Logical Block
$Ø1	Write Logical Block
$Ø2	Write/Verify Logical Block

The three Profile commands behave in exactly the same fashion as do the corresponding instructions on ProFile, with one small exception: the Read Logical block command does not include information concerning Retry count or Sparing threshold (however, because of a side effect in the way that the Host/Controller interface was designed, the Host may write as many command bytes to the controller as it chooses. The controller will only decode the first 4.). The form of each command is:

<$ØØ | $Ø1 | $Ø2> < 3 Bytes of Logical Block Address >

There are two 'special' logical addresses defined in the ProFile protocol, namely $FFFFFF (-1) and $FFFFFE (-2). Logical address < -1 > returns as it's value Device_ID (as explained under the Widget Diagnostic commands) and Logical address < -2 > returns as it's value Widget's spare table structure in it's raw form. It should be noted that if at any time Widget can not pass it's self test that it will refuse to communicate via logical commands (both ProFile and System type commands). Widget will respond to Diagnostic commands at all times, however.

The rest of the commands available on Widget are a complete departure from the ProFile way of doing things. The new form of command is:

(< Command_Byte >
< Instruction_Byte >
[Instruction_Parameter_String]
< CheckByte >)

Command_Byte ::= < CommandType_Nibble + CommandLength_Nibble >

CommandType_Nibble ::= < Diagnostic_Command | System_Command >

Diagnostic_Command ::= < $13 >

WD26 Program

System_Command ::= < $20 >

CommandLength_Nibble ::= Count of all bytes in the command string NOT including the first one. This length is used only to calculate the checkbyte, and not to parse the command, therefore there is a large variety of commands that perform exactly the same function but differ in format in that their lengths are not the same.

IF System_Command
THEN Instruction_Byte ::= <Sys_Read | Sys_Write | Sys_WrVer>

IF Diagnostic_Command
THEN Instruction_Byte ::= <
Read_ID |
Read_Controller_Status |
Read_Servo_Status |
Send_Servo_Command |
Send_Seek |
Send_Restore |
Set_Recovery |
Soft_Reset |
Send_Park |
Diag_Read |
Diag_ReadHeader |
Diag_Write |
Store_Map |
Read_SpareTable |
Write_SpareTable |
Format_Track |
Initialize_SpareTable |
Read_Abort_Stat |
Reset_Servo |
Scan >

Instruction_Parameter_String ::= (This string is instruction dependent, and will be formally specified at the same time as the individual instructions.)

CheckByte ::= (This byte is the ones-complement of the sum, in MOD-256 arithmetic, of all the bytes including the Command_Byte).

DIAGNOSTIC_COMMANDS

Widget's "personality", or the manner in which it behaves, can be thought of as having two distinct parts: 1) that portion that is dictated by the hardware and 2) that portion that is controlled by the firmware. As trite as that last statement may seem on the surface, the fact remains that the part of Widget that is the hardware is not easily molded to adapt to different environments. The same is true, but not quite in the same manner, for the firmware - the code is locked in a ROM of some sort and costs a lot to change. How then can Widget's "personality" be changed (on-the-fly) to "adapt" to a new environment? The answer in this case to architect the firmware in a layered fashion: build the intelligence required to run Widget in it's normal operating mode from a pool of discrete, primitive functions; these primitive functions in most cases have only one particular task that they are capable of completing. The implication of this architecture is that with very little effort these same primitive functions are availble to the host system, and thus make Widget a little "Schizoid". Such luxuries do not come without their hidden costs, however. For one thing, the Widget controller is slightly more expensive to manufacture (a cost that I believe pales in the sight of the added test/diagnostic capabilities) because of the additional code space required for all the bells and whistles, and another is that someone must now develop Host software to emulate the controller firmware design of choice.

The purpose of the rest of this section on Diagnostic Commands is to aquaint the casual/not-so-casual designer of Host software as to how to make the best use of Widget's multiple personality capabilities.

Read_ID ::= < $ØØ >

Instruction_Parameter_String ::= NULL
This diagnostic command requires Widget to deliver to the host some device specific information. The structural layout of the data returned is:

STRUCTURE Identitiy_Block

(this identity block is defined by the data structures contained within it; you will note, however, that a comment is given explaining the type of structure for a given element and range of bytes (if the entire structure is thought of as a linear array of bytes) that include the structure. An example is NameString (first element to be defined below) which is a 13-character ascii string, and is located in bytes $Ø thru $C of the returned block.

```
NameString ::= <
                10MB_Name |
                20MB_Name |
                40MB_Name   ( 13 Bytes/$ØØ:$ØC; Ascii String )>
      10MB_Name ::= < 'Widget-10     ' >
      20MB_Name ::= < 'Widget-20     ' >
      40MB_Name ::= < 'Widget-40     ' >

      DeviceType ::= <Device.Widget+Widget.Size+Widget.Type ( 3 Bytes/$Ø
:$ØF )>
      Device.Widget ::= <$ØØØ1 ( 2 Bytes/$ØD:$ØE )>
      Widget.Size ::= <Size_1Ø | Size_2Ø | Size_4Ø ( 1 Nibble, Byte $ØF/
its 7:4 )>
           Size_1Ø ::= <$ØØ>
           Size_2Ø ::= <$1Ø>
           Size_4Ø ::= <$2Ø>
      Widget.Type ::= <System | Diagnostic ( 1 Nibble, Byte $ØF/bits 3:Ø
)>
           System ::= <$ØØ>; This refers to the type of firmware that is
imbedded in
           Widget.
           System firmware will not allow the host to Format, or
           Initialize_SpareTable; Diagnostic firmware will.

           Diagnostic ::= <$Ø1>

Firmware_Revision ::= <( 2 Bytes/$1Ø:$11 )>

Capacity ::= <Cap_1Ø | Cap_2Ø | Cap_4Ø ( 3 Bytes/$12:$14 )>
     Cap_1Ø ::= <$ØØ4CØØ>
     Cap_2Ø ::= <$ØØ98ØØ>
     Cap_4Ø ::= <$Ø13ØØØ>

Bytes_Per_Block ::= < $Ø214 ( 2 Bytes/$15:16 )>
```

```
        Cyl_1Ø ::= <$Ø2Ø2>
        Cyl_2Ø ::= <$Ø2Ø2>
        Cyl_4Ø ::= <$Ø4Ø4>

    Number_Of_Heads ::= <$Ø2 ( 1 Byte/$19 )>

    Number_Of_Sectors ::= < Sctr_1Ø | Sctr_2Ø | Sctr_4Ø ( 1 Byte/$1A )>
        Sctr_1Ø ::= <$13>
        Sctr_2Ø ::= <$26>
        Sctr_4Ø ::= <$26>

    Number_Of_Possible_Spare_Locations ::= <$ØØØØ4C ( 3 Bytes/$1B:$1D )>

    Number_Of_Spared_Blocks ::= << 3 Bytes/$1E:$2Ø; range Ø..$4B )>

    Number_Of_Bad_Blocks ::= << 3 Bytes/$21:$23; range Ø..$4B )>
```

Read_Controller_Status ::= <$Ø1>

Every time an operation completes (either successfully or exceptionally) Widget will return what I refer to as Standard_Status, thus allowing the Host system an opportunity to change it's flow of execution based on state of the Status. Normally, this Standard_Status is all that is necessary to ensure continuous operation. In the exceptional case, or when the Host system is emulating the controller's functions, additional information concerning the state of Widget is mandatory: without it the Host simply could not make an optimum choice in deciding a course of action.

Controller_Status is then a means for the Host system to interrogate Widget further. Each Status (with the exception of Abort_Status, which is a seperate command and is discussed later in this document) belongs to a homogeneous data structure: namely a four byte quantity containing a bit map representing the various exceptional conditions (active high) that is available as the first four bytes read from the controller upon completion of the current command.

There are eight status' available to the Host system. The Host requests a specific status by setting Instruction_Parameter_String to the value corresponding to the status needed.

```
IF ( Instruction_Byte = Read_Controller_Status )
          THEN Instruction_Parameter_String ::= <
                              Standard_Status |
                              Last_Logical_Block |
                              Current_Seek_Address |
                              Current_Cylinder |
                              Internal_Status |
                              State_Registers |
                              Exception_Registers |
                              Last_Seek_Address >
```

The four byte response to each of the above status requests is of the form:

Result ::= < ByteØ Byte1 Byte2 Byte3 >

```
Standard_Status ::= <$ØØ>

     Byte0 ::= <
                 Bit7: Other than $55 response from Host
                 Bit6: Write Buffer OverFlow
                 Bit5: ( not used )
                 Bit4: ( not used )
                 Bit3: Read error
                 Bit2: No matching header found
                 Bit1: Unrecoverable servo error
                 Bit0: Operation Failed >

     Byte1 ::= <
                 Bit7: ( not used )
                 Bit6: Spare Table OverFlow
                 Bit5: 5 or less spare blocks available
                 Bit4: ( not used )
                 Bit3: Controller SelfTest failure
                 Bit2: SpareTable has been updated
                 Bit1: Seek to wrong track occured
                 Bit0: Controller aborted last operation >

     Byte2 ::= <
                 Bit7: First status response since power-on reset
                 Bit6: Last Logical_Block_Number was out of range
                 bit5:0 ( not used ) >

     Byte3 ::= <
     Bit7: Read Error detected by ECC circuitry
     Bit6: Read Error detected by CRC circuitry
     Bit5: Header Timeout on last read
     Bit4: ( not used )
     Bit3:0  : number of unsuccessful retries ( out of 10 ) for last
     read
```

Last_Logical_Block ::= < $Ø1 >

Byte0 ::= < $ØØ >

Byte1 ::= < (Most Significant Byte of Logical_Block_Number) >

Byte2 ::= < (Middle Byte of Logical_Block_Number) >

Byte3 ::= < (Least Significant Byte of Logical_Block_Number) >

Current_Seek_Address ::= < $Ø2 >

Byte0 ::= < Most Significant Cylinder Address >

Byte1 ::= < Least Significant Cylinder Address >

Byte2 ::= < Head Address >

Byte3 ::= < Sector Address >

Current_Cylinder ::= < $Ø3 >

(The Current_Cylinder differs from the Current_Seek_Address in that it is perfectly reasonable for the Servo to have placed the heads on another track under certain circumstances; for example, the drive may have been bumped)

Byte0 ::= < Most Significant Cylinder address >

Byte1 ::= < Least Significant Cylinder address >

Byte2 ::= < Most Significant Cylinder of current seek address >

Byte3 ::= < Least Significant Cylinder of current seek address >

```
Internal_Status ::= < $Ø4 >

     Byte0 ::= < ( Register: Excpt_Status )
                 Bit7: Recovery ( active high --> Recovery ON )
                 Bit6: Spare almost full
                 Bit5: Buffer structure is contaminated
                 Bit4: Power reset has just occured
                 Bit3: Current Standard Status is non-zero
                 Bit2:1 : ( not used := Ø )
                 BitØ: Set if controller LED is lit >

     Byte1 ::= < ( Register: DiskStatus )
                 Bit7: On_Track ( heads are position where they
                 should be )
                 Bit6: Read a Header after Recal
                 Bit5: current operation is a WRITE operation
                 Bit4: Heads are parked
                 Bit3:  Do sequential search of Logical Block
                 look-ahead structure
                 Bit2: Last commad was a multiblock command
                 Bit1: Seek_complete
                 BitØ: Servo offset ( auto ) is on >

     Byte2 ::= < ( Register: BlkStatus )
          This byte of status is valid ONLY after a ProFile/System
          command. If the byte is read after a Diagnostic command it
          will contain information concerning the last
          non-Diagnostic command.
                 Bit7: SeekNeeded ( a seek was needed to arrive at
                 the current block )
                 Bit6: Head_Change_Needed ( like Bit7, but Head
                 change instead of seek )
                 Bit5:2 $ØØ ( not used )
                 Bit1: Current Block is a Bad Block
                 BitØ: Current Block is a Spare Block >

     Byte3 ::= < $ØØ ( not used ) >
```

```
State_Registers ::= < $05 >

     Byte0 ::= < $00 ( not used ) >

     Byte1 ::= < ( Register: SelfTst_Result )
                 Bit7: Ram_Failure
                 Bit6: Eprom_Failure
                 Bit5: Disk_Speed_Failure
                 Bit4: Servo_Failure
                 Bit3: Sector_Count_Failure
                 Bit2: State_Machine_Failure
                 Bit1: Read_Write_Failure
                 Bit0: No_Spare_Table_Found >

     Byte2 ::= < ( Register: Port2 )
                 Bit7: Disk Read/Write direction set to Read
                 Bit6: Servo is able to accept a command ( SioRdy )
                 Bit5: MSel1  ( MSel0  and  1 determine  the  memory
                 source and destination )
                 Bit4: Msel0
                 Bit3: BSY
                 Bit2: CMD
                 Bit1: Ecc Error
                 Bit0: State machine is running >

     Byte3 ::= < ( Register: Controller_Status_Port )
                 Bit7: CrcError ( active low )
                         (   this bit   is valid   ONLY when the
                         controller state machine  is NOT in reset,
                         which should be every time that this bit is
                         read by the host. Therefore, if this status
                         bit  indicates  a CrcError,  then something
                         has croaked. The normal way for the host to
                         check if a Crc or Ecc error has occured  is
                         to   examine   Status: Exception_Registers
                         which are dicussed below. )

                 Bit6: Write_Not_Valid ( active low )
                         ( as in  CrcError, this bit  is valid  only
                         when the state machine is NOT in reset. The
                         information   expressed  by   this  bit  is
                         converted into a type of ServoError,  which
                         is found in Status: Exception_Registers. )

                 Bit5: ServoReady
                 Bit4: ServoError
                         (  the servo  status bits  listed above are
                         further explained  in  Appendix A:    Servo
                         Processor  Documentation.  Essentially  the
                         two bits  combine  to  form  four  possible
```

servo states; the normal condition is ServoReady AND (NOT ServoError). }

Bit3:Ø Current controller state-machine state. { as in CrcError and Write_Not_Valid, these status bits are valid only when the state machine is NOT in reset, and should read $ØØ any other time. }

On the surface it appears that this byte is of limited use for non real- time situations. It is, however, invaluable in trying to decide if the Servo Processor is healthy, wealthy, and wise. It also provides a means for diagnosing a sick state machine.

Exception_Registers ::= < $Ø6 >

ByteØ ::= < (Register: RdStat)
Bit7: Read error occured on last read attempt
Bit6: Servo Error while reading
Bit5: At least one successful read in last read attempt (this means that valid data is residing in Buffer2)
Bit4: No matching header was found during last read attempt
Bit3: CrcError OR EccError occured during last read attempt
Bit2:Ø $ØØ (not used) >

(a read attempt is defined as being the sequence of events normally associated with reading a single block of data. In the case where the first read of a block was invalid for some reason, AND Recovery is active, then the controller will automatically retry 9 times: 1Ø tries total. For example, if the first read was invalid because of a CrcError, but the second thru tenth reads are all correct then the status bits that will be active are Bit5, and Bit3. Correct and valid data will be both in the normal Read buffer and in Buffer2.)

Byte1 ::= <
Bit7: Error detected by ECC circuitry
Bit6: Error detected by CRC circuitry
Bit5: Header timeout
Bit4: (noy used := Ø)
Bit3:Ø : Number of bad retries during last read attempt >

(For the above example, this status byte will contain the value $C1)

Byte2 ::= < (Register: WrStat)
Bit7: Write error occured on last write attempt
Bit6: Servo Error while writing
Bit5: At least one successful write during last write attempt
Bit4: No matching header found during last write attempt
Bit3:Ø-$ØØ (not used)

(A write attempt is much the same as a read attempt in that there are several events that can keep the controller from writing a block successfully - and can be detected at the time of the attempted write. If Recovery is active then the controller will first copy the write buffer to Buffer2 and then retry)

Byte3 ::= < Number of bad retries during last write attempt >

Read_Servo_Status ::= < $Ø2 >

Instruction_Parameter_String ::= < Ø..8 >

This status command is used to interrogate the Servo Processor in much the same way that Read_Controller_Status is used. In fact, the form of the result is the same four byte bit-mapped quantity.

This command is of particular value to a diagnostician that is interested in 'picking-about' with the servo processor without dismantling Widget as a subsystem. Refer to Appendix A: Servo Processor Documentation for a complete description of the various status' available and their resulting bit descriptions.

Send_Servo_Command ::= < $Ø3 >

Instruction_Parameter_String ::= < ByteØ Byte1 Byte2 Byte3 >

Normally, the Host will allow the controller to manipulate the servo processor in order to perform useful (or maybe not so useful!) work. For example, let's suppose that the Host system wishes to move the disk drive heads from one track to another. Under normal operating conditions the preferred way to perform this task is to use the Send_Seek command (explained below). However, the Host has the capability to bypass the controller and direct the servo processor. Indeed, the Host can issue the servo command to position the heads (via the Send_Servo_Command) so that the seek is completly transparent to the controller. The implication of this command is that the Host can gain even more control of the system if it so chooses.

A more complete description of the Servo Commands can be read in Appendix A: Servo Processor Documentation.

ByteØ ::= < S_Command + S_Direction + Hi_Magnitude >
S_Command ::= <
Offset
Diagnostic
DataRecal
FormatRecal
Access
Access_Offset
Home

Offset ::= < $1Ø >

The Offset command allows the Host to microstep the heads in either a positive or negative direction from the center of the track. The Widget Firmware does not make use of this feature! I have instead left this to a more specific data recovery program that is run by the Host. The value

and direction of the microstep are sent to the Servo Processor in Byte2.

Diagnostic ::= < $2Ø { this command is not implemented in the Servo } >

DataRecal ::= < $4Ø >

DataRecal { and also FormatRecal } is used as a 'Get the servo in a known state' command, and is usually sent by the controller during initialization time or whenever the servo is not 'Ready'. This command places the heads over the first data track closest to the inside diameter of the disk, within a tolerance of 3 tracks. The accepted method for making certain that the heads are over a known track following a DataRecal is to read a header and use the track information located in the header to establish the location.

FormatRecal ::= < $7Ø >

This command is identical to the DataRecal command except for the track that the heads end up over upon completion: about 36 tracks closer to the inside diameter of the disk. Unlike the DataRecal command, however, the disk surface in this area is not likely correctly store information written there. This command then is used to supply an absolute reference point when formating the drive.

Access ::= < $8Ø >

I use the term 'access' and 'seek' interchangeably within the context of this document. The servo Access command is used to position the heads a relative distance from their current position. The Servo Processor has no knowledge concerning absolute position and it is up to the controller { real or emulated } to supply the relative distance. This information is passed along in ByteØ and Byte1.

Access_Offset ::= < $9Ø >

The difference between an Access and an Access_Offset is that the assumption is made that heads will position themselves within a 'tolerable' distance of the center of the track with an Access command, while no such assumption is made with an Access_Offset command. There is some information written on each track of the disk 'under' the index mark. This information is used by the servo processor to 'calculate' the center of the track { data center } and position the heads accordingly. Because the servo must wait for the index to arrive under the heads before it can read this information there is an implied latency of about 1 revolution { currently 19.4 msec }

attached to each Access_Offset. Normally, the Widget controller will use the Access command for all reads, and the Access_Offset command for all writes.

Home ::= < $CØ >
When the heads are 'Homed' they are sent completely off the data surface and held in position very near the inside diameter crash stop.

S_Direction ::= < Positive | Negative >

Positive ::= < $Ø4 (move the heads toward the outside diameter) >
Negative ::= < $ØØ (move the heads toward the inside diameter) >

Hi_Magnitude ::= < Ø..3 (move the heads a multiple of 256 tracks) >

Byte1 ::= < Low_Magnitude ::= Ø..255 >
Hi_Maginitude + Low_Maginitude, and S_Direction establish the relative distance the heads must move to arrive at the target track.

Byte2 ::= < Offset_Direction + Auto_Offset_Switch + Offset_Magnitude >

This command byte, when used with the Offset command, establishes the degree and direction of microstepping.

Offset_Direction ::= < Positive | Negative >
Positive ::= < $8Ø (offset towards outside diameter) >
Negative ::= < $ØØ (offset towards inside diameter) >

Auto_Offset_Switch ::= < ON | OFF >
ON ::= < $4Ø (turn automatic track centering on without an access command) > OFF ::= < $ØØ (do not auto track center on this command) >

Offset_Maginitude ::= < Ø..32 >

Byte3 ::= < Baud_Rate + Power_On_Reset >

Baud_Rate ::= < 19.5k_Baud | 57.6k_Baud >

The servo 'comes up' at 19.5k baud because of the test equipment used on it before it is integrated into a system. Once it is running with a controller, however, it is run continuosly at 57.6k baud. This parameter is also a bit misleading in that once the servo has been told to go to 57.6k it will forever more ignore this parameter: in other words it is impossible to go from the higher baud rate to the lower without reseting the servo processor.

19.5k_Baud ::= < $ØØ >
57.6_Baud ::= < $8Ø >

Power_On_Reset ::= < $4Ø >
This is one of three way to reset the servo processor (such variety!). The other two are: 1) Power switch, and 2) have the controller pull on the servo reset line. Out of all three methods, choice two is the most preferable in that the controller will completely initialize all the drive parameters related to the servo as well as automatically go to the higher baud rate.

Send_Seek ::= < $Ø4 >

Instruction_Parameter_String ::= < HiCyl LoCyl Head Sector >

Widget's Send_Seek command allows the Host system to place the heads over any track on the disk. The value of the seek address sent in the parameter string is used read/write a block of data using the diagnostic commands for those functions. For example, for the Host to read Cylinder 1, Head Ø, Sector 18 a Seek_Command would be issued for that combination of cylinder, head, and sector ($ØØØ1 ØØ 12) followed by a Diag_Read (explained below).

Send_Restore ::= < $Ø5 >

Restore_Data ::= < $4Ø >
Restore_Format ::= < $7Ø >

The Send_Restore command is used by the host to initialize the servo processor and to put the heads in a known location. This command is the same as performing a Data/Format Recal except that the controller updates it's internal state to account for the new servo position (as opposed to using the Send_Servo_Command, which is transparent to the controller).

Set_Recovery ::= < $Ø6 >

Instruction_Parameter_String ::= < ON | OFF >

ON ::= < $Ø1 >
OFF ::= < $ØØ >

To the best of my ability I have attempted to make the exception handling characteristics of Widget a binary set: either Widget handles everything, or the Host system does. The command Set_Recovery is the Host's link with this all or nothing world in that it is through this instruction that the Host can gain control of the media. When Widget comes up after being reset it assumes control and sets Recovery to be ON. The Host system must overtly change this state (via Set_Recovery) if it wishes to emulate a different exception handling criteria. Once Recovery is OFF, the controller will always fail in an operation if an exception occurs: the Host system MUST assume responsibility for ALL error handling.

Soft_Reset ::= < $07 >

Instruction_Parameter_String ::= < NULL >

This commands instructs the Widget firmware to restart it's flow of execution at it's initialization point. The results should be the same (from a software point-of-view) as a power-reset.

Send_Park ::= < $Ø8 >

Instruction_Parameter_String ::= < NULL >

When the Host issues a Send_Park command to the controller the results are that that the heads are moved off the data surface and held very near the inside diameter crash stop. The difference between this command and the Send_Servo_Command: Home is that Home is performed 'open-loop' with the crach stop as it's reference point, while Send_Park is an access command to a specific track. The net result is a fairly hefty saving of time: the access command can be an order of magnitude quicker than Home/Recal.

Diag_Read ::= < $Ø9 >

Instruction_Parameter_String ::= < NULL >

The Diag_Read command is used to read the block on the disk pointed to by the last seek address. This instruction is valid for states that the controller might be in: it is advised that a Send_Seek command precede the Read. The form of the returned data is exactly the same as that of ProFile_Read or a Sys_Read in that 4 bytes of Standard_Status precede the block of data.

Diag_ReadHeader ::= < $ØA >

Instruction_Parameter_String ::= < Sector ($Ø..$12) >

When the heads are positioned over an unknown location, or when it is suspected that a block's header is shot, it is time to use the Diag_ReadHeader command. This instruction allows the host to 'suck-up' both whatever information is residing in the block's header field as well as the data from that block. The form of the result is:

Result ::= (
< Standard_Status/$ØØ:$Ø3 >
< Header/$Ø4:$Ø9 >
< Gap/$ØA:$ØF >
< Sync/$1Ø:$11 >
< Data/$12.. >)

Standard_Status ::= < (as defined above) >

Header ::= < HiCyl LoCyl HdSct -HiCyl -LoCyl -HdSct >

HiCyl ::= < 1 Byte, Most significant cylinder address >

LoCyl ::= < 1 Byte, Least significant cylinder address >

HdSct ::= < 1 Byte, bits7:6 are head address, bits5:Ø are sector address >

-HiCyl ::= < Ones-complement of HiCyl >

-LoCyl ::= < Ones-complement of LoCyl >

-HdSct ::= < Ones-complement of hdSct >

Gap ::= < 5 bytes of $ØØ >

Sync ::= < $Ø1ØØ >

Diag_Write ::= < $0B >

Instruction_Parameter_String ::= < NULL >

This instruction allows the host to write a block of data to the location on disk pointed to by the last seek address. Diag_Write is valid for all states that the controller may wind up in, but it is recommended that a Send_Seek command precede the write command to ensure that the correct block will be written.

Store_Map ::= < $0C >

Instruction_Parameter_String ::= < NULL >

The Store_Map command is to be used by the Host to logically re-interleave Widget. Widget will be used on a number of target hosts, each of which would like to optimize the performance (sequential) of the disk drive. This optomization can occur in one of two ways: 1) either seperate lines are set up in manufacturing to initialize Widgets specifically for each target host or 2) we can manufacture a single Widget unit and have the Host initialize the drive for it's specific requirements.

Included in the SpareTable structure is a data structure called the InterLeave_Map. This map is used as another level of logical addressing during the calculation of a cylinder, head, and sector address from a given logical block address. Specifically stated, once a sector address has been determinied it is used as an index into the InterLeave_Map and a new sector address is generated (the InterLeave_Map is an array with the same number of entries as there are sectors, and each entry must be unique and valued within the range of legal sector values).

It is extremely important that the host system proceed with caution when changing the Map. A remapping of the elements within the SpareTable is REQUIRED with every change to the Map (this is because as the sectors are logically remapped the defects that stay with a physical address move around relative to a logical block's number). For this reason I suggest that all changes to the map be done using the Write_SpareTable command in conjunction with a remapping of all the spare/bad blocks.

This command is externally executed (by the host) as a write command. The first Number_Of_Sectors worth of data in the buffer are assumed to be the new map.

Read_SpareTable ::= < $ØD >

Instruction_Parameter_String ::= < NULL >

Reading (and writing) Widget's spare table is an absolute must for diagnostic purposes, and if the Host wishes to emulate the controller. The result of this instruction is identical to performing a ProFile_Read from block $FFFFFE and has the form:

```
Result ::= (
     < Standard_Status/$ØØ:$Ø3 ( as defined above ) >
     < Fence/$Ø4:$Ø7 >
     < RunNumber/$Ø8:$ØB >
     < Format_Offset/$ØC >
     < Format_InterLeave/$ØD >
     < HeadPtr_Array/$ØE:$8D >
     < SpareCount/$8E >
     < BadBlockCount/$8F >
     < BitMap/$8A:$93 >
     < Heap/$94:$1C3 >
     < InterLeave_Map/$1C4:$1D7 >
     < CheckSum/$1D8:$1D9 >
     < Fence/$1DA:$1DD > )
```

Fence ::= (< $FØ > < $78 > < $3C > < $1E >)

RunNumber ::= < 32-bit interger >

The RunNumber is incremented each time the spare table is writen to the disk. Because two copies are kept on the disk, the RunNumber is used to decide which is the more recent of the two should both copies of the table not be updated.

Format_Offset ::= < $ØØ..NumberOfSectors >

Format_Offset is the number of physical sectors there are from index mark until logical sector Ø.

Format_InterLeave ::= < $ØØ..$Ø6 >

This number is the interleave factor for this disk and is used in calculating where each of the logical sectors are in terms of actual physical sectors.

HeadPtr_Array ::= < ARRAY[Ø..127] of HeadPtr >

HeadPtr ::= < Nil + Ptr >

Nil ::= < $ØØ | $8Ø >

If a HeadPtr is Nil, then there is no linked-list structure in the heap corresponding to the current logical block number.

Ptr ::= < $ØØ..$7F >

A Ptr is a seven bit data structure that 'points' to a specific location within the Heap (if the Heap can be though of as a linear array of bytes, the a Ptr is used to index into that array). To arrive at the actual index value within the Heap, the Ptr must first be multiplied by four.

When a disk is formatted and fresh data is being written to it, each logical block is asigned the first available physical block on the disk. Therefore you would expect that LogicalBlock(Ø) would occupy PhysicalBlock(Ø), L(1) --> P(1), etc. There are instances, however, when a block of data must be relocated to another space on the disk that does not follow the original progression (for example, the original space was defective). In order to 'find' these relocated blocks in the future a record must be kept as to where all these relocated blocks have been put. This record takes the form of 128 linked lists having the form HeadPtr(n) --> LinkedList(n), where n := Ø..127. The algorithm for deciding whether or not a LogicalBlock has been relocated is to extract bits 16:1Ø from the LogicalBlockNumber and use it as an index into the HeadPtr_Array. If the HeadPtr associated with this index value is Nil then LogicalBlock has not been relocated else use HeadPtr.Ptr to search the linked list corresponding to this HeadPtr value. Now to decide if the LogicalBlock has been relocated a test must be made as the linked list is traversed by comparing the LogicalBlockNumber's bits 9:Ø to the current list element's token value. If they match then LogicalBlock has been relocated and it's new position is a multiple of the list element's position in the Heap.

```
SpareCount ::= < $ØØ..$4C >
BadBlockCount ::= < $ØØ..$4C >

BitMap ::= < ARRAY[ Ø..$4B ] of Bits >
    . The bit map is used to keep a record of which spare blocks
      are occupied, and their locations on the disk.

Heap ::= < ARRAY[ Ø..$4B ] of ListElement >

     ListElement ::= (
               < Nil+Used+Useable+Spr_Type+Data_Type >
               < Token >
               < Ptr > )

          Nil ::= < $8Ø ( IF Nil THEN End_Of_Chain ) >
          Used ::= < $4Ø >
          Useable ::= < $2Ø >
          SprType ::= < Spare | BadBlock >
```

```
            Spare ::= < $1Ø >
            BadBlock ::= < $ØØ >
      Data_Type ::= < Data | SpareTable >
            Data ::= < $Ø2 >
            SpareTable ::= < $Ø8 >

      Token ::= < Bits9:Ø of the LogicalBlockNumber >

InterLeave_Map ::= < ARRAY [Ø..NbrSctrs] OF Ø..NbrSctrs >

CheckSum ::= < the sum of all bytes in the spare table from the
first fence to the end of the heap, in MOD-65536 arithmetic >
```

Write_Spare_Table ::= < $ØE >

Instruction_Parameter_String ::= (< $FØ > < $78 > < $3C > < $1E >)

This command allows the Host to 'force' a new spare table on the controller, and is executed just like any of the other write commands (the data in this case MUST conform to the structure presented in Read_SpareTable). The data sent to the controller is written to the two spare table locations on the disk.

Format_Track ::= < $ØF >

Instruction_Parameter_String ::= <
< Format_Offset >
< Format_InterLeave >
< PassWord >

Format_Offset ::= < $ØØ..Number_Of_Sectors >
This parameter dictates which sector (beginning with sectorØ - the first physical sector after index mark) will be logical sector Ø for that track.

Format_InterLeave ::= < $ØØ..$Ø6 (interleave factor) >

PassWord ::= (< $FØ > < $78 > < $3C > < $1E >)

The format command is used to:

1. Operate on the track that is currently beneath the heads - this implies that the Host had best perform a Send_Seek command prior to formatting a track.

2. AC erase the entire track - this implies that all data stored on this track has acheived Nirvana and are living happlily ever after in the great bit bucket in the sky.

3. New headers will be layed down on EVERY sector of the track.

```
Initialize_SpareTable ::= < $1Ø >

Instruction_Parameter_String ::= (
                                    < Format_Offset >
                                    < Format_InterLeave >
                                    < PassWord >

     Format_Offset ::= < $ØØ..Number_Of_Sectors >

     Format_InterLeave ::= < $ØØ..$Ø6 ( interleave factor ) >

     PassWord ::= ( < $FØ > < $78 > < $3C > < $1E > )
     This  command form the  Host instructs the  controller to 'wipe
     the slate clean'  as far  as the SpareTable  is concerned.  The
     initialized table is written to disk.
```

Read_Abort_Status ::= < $11 >

Instruction_Parameter_String ::= < NULL >

Read_Abort_Status will return valid data only AFTER the controller has aborted (identified by Standard_Status.Byte1.BitØ). The form of the result is a sixteen byte string, and the contents are the contents of the controller's registers at the time of the abort - with the exception of bytes $ØE and $ØF, which constitute the return address of the procedure that called the Abort routine. Because all of the information that can be derived from this request from is extremely firmware dependent an appendix (Appendix C: Abort_Status Variables) has been created that hopefully will be updated with each firmware release.

Reset_Servo ::= < $12 >

Instruction_Parameter_String ::= < NULL >

Reset_Servo allows the host to initialize the servo processor without having to power the device down. The controller will automatically reset the Servo, check for valid initial conditions and perform a Data_Recal.

Scan ::= < $13 >

Instruction_Parameter_String ::= < NULL >

The Scan command causes the Widget to read all blocks that are with the range of blocks set aside for user data blocks. If any of these blocks are bad then the block will either be relocated (if the data can be recovered) or marked as bad and relocated on the next write to that block. The SpareTable can be examined before and after a Scan command find the locations of all bad blocks.

SYSTEM_COMMANDS

System_Commands have been implemented for essentially two reasons:

1. I felt that it was important for Widget to add one more check on the CMD/BSY handshake: namely the addition of a checkbyte following the command string.

2. In order to increase the performance of the system without modifying the hardware it was critical to introduce another level of parallelism into the Host/Controller interface. Most (60% or greater) of the reads for a specific block on the disk are followed by a read for the logically sequential block. In fact, in the extreme case of Lisa, this percentage is almost 100%. Therefore I have suppressed the command decoding for all but the first block read (over a small range). The implementation, then, for this added parallelism is to send an additional parameter with the (first) LogicalBlock indicating the number of blocks to be read.

This implementation holds for Reads and Writes, but not for WriteVerifies. I have taken the liberty of assuming (hopefully correctly) that WriteVerifies do not exhibit the same characteristic behaviour as the other two types of commands, and that they are fairly long commands to begin with. The trade-off then was one of saving code space (a Sys_WrVer is the same routine as a Pro_WrVer, but with command checkbyting) vs. adding a third multiblock function with limited performance increases.

The protocol for System commands is slightly different then that of Profile commands. In the case of a Read command, each block of data is transfered to the host when it received by the controller: there is NO buffering of disk blocks on Widget at this time. The transfer looks just like other read-style transfers in that Standard_Status is sent with the data block and the data block is the same length (532 bytes). Instead, however, of responding with the basic 'Controller is ready for command' response when the Host sets CMD (after storing the data block) the controller will respond with a 'Controller ready to get next block' response.

Sys_Read ::= < $00 >

Instruction_Parameter_String ::= (< Block_Count > < LogicalBlock >)

Block_Count ::= < $01..$13 >
This parameter is the number of blocks to be read that follow sequentially from LogicalBlock. It is assumed that one block (LogicalBlock) will be read, making the Block_Count the number of blocks following the first one that is to be read, also.

LogicalBlock ::= < L_10MB | L_20MB | L_40MB >
L_10MB ::= < $000000..$004BFF >
L_20MB ::= < $000000..$0097FF >
L_40MB ::= < $000000..$012FFF >

```
Sys_Write ::= < $01 >

Instruction_Parameter_String ::= < < Block_Count > < LogicalBlock  >
>

     Block_Count ::= < $01..$13 >

     LogicalBlock ::= < L_10MB | L_20MB | L_40MB >
          L_10MB ::= < $000000..$0048FF >
          L_20MB ::= < $000000..$0097FF >
          L_40MB ::= < $000000..$012FFF >


Sys_WrVer ::= < $02 >

Instruction_Parameter_String ::= < LogicalBlock >

     LogicalBlock ::= < L_10MB | L_20MB | L_40MB >
          L_10MB ::= < $000000..$0048FF >
          L_20MB ::= < $000000..$0097FF >
          L_40MB ::= < $000000..$012FFF >
```

HANDSHAKE PROTOCOL

Both Widget and ProFile share the same Host interface scheme, and therefore a lot in common when it comes to trying to communicate with the Host system. ProFile's protocol is documented in 'ProFile Communication Protocol', and a follow-up document titled 'The Extended ProFile Protocol' written by Karl Young is available for more detail.

The actual sequence of events can be portayed as follows:

```
Protocol_Sequence ::= (
                    < Initial_HandShake >
                    < Command_DownLoad >
                    < Response_HandShake >
                    [ Data_Received_HandShake ]
                    < Final_HandShake > )
```

Initial_HandShake ::=

1. Host asserts CMD, sets data direction to read
2. Controller asynchronously responds by:
   a. Writing $01 to the Host
   b. Asserting BSY

3. If the Host recognizes the controller response, it will respond by:
   a. Writing a $55 to the controller
   b. Otherwise it will write a $AA
   c. In either case the Host will de-assert CMD.

4. The controller will respond to the Host by:
   a. In either case (whether the Host responded with a $55 or $AA or anything else) the controller will eventually end up waiting for the next instance of CMD.
   b. If the response was a $55 then the controller will be a 'captive' audience, anxiously awaiting instructions from the Host as to what to do next.
   c. Otherwise, the controller will Abort, and leave Standard Status saying so in it's buffer where the host can read it. The state of the command sequence for the controller then becomes Initial_HandShake, and the Host should read do it's best to read the Standard Status as soon as it notices that the handshake sequence has been changed. The execption to this 'Otherwise' is when the response from the Host is a FreeProcess response (explained below).

Command_DownLoad ::=

1. The Host writes a variable length string of hex bytes to the controller. The address of where these bytes are sent is set up by the controller in the Initial_HandShake phase. The length of the hex string is up to the Host, but is intended to be the length of a command string (indeed, the string of bytes is supposed to be a command string!). The controller knows to increment it's address counter (remember, it is responsible for loading the string into it's memory) by a falling edge of STROBE from the interface card.

Response_HandShake ::=

1. The Host asserts CMD

2. The controller responds asynchronously by first reading it's buffer in the locations that it set aside for the Host to perform it's command download, doing what is necessary to decode the command (i.e., validating the checkbyte, making certain that the command was of the right type, and decoding the command). It then writes a response byte to the Host which has the value of (Instruction_Byte + 2).

3. The controller asserts BSY

4. (look at 3. for Initial_HandShake)

5. If the controller receives a $55 then it will continue executing the command, otherwise it will Abort and return to Initial_HandShake.

Data_Received_HandShake ::=

1. If the controller is expecting data (as is the case for a write command) then in the Response_HandShake it will de-assert BSY and wait for the next occurance of CMD.

2. When the Host 'sees' BSY become de-asserted it will then write as much data as it pleases (like the command download, the controller dictates the address of the data while the Host dictates the length).

3. The Host the asserts CMD

4. The controller responds asynchronously to the Host by writing a $06 to the Host.

5. The controller then asserts BSY

6. Assuming the Host accepts the response from the controller, it will respond by writing $55 back to the controller and then de-asserting CMD.

7. The controller will then continue executing the command.

Final_HandShake ::=

1. When the controller finishes with the execution of the instruction,
it will put the latest Standard_Status in a location in it's buffer
where it will be accessible to the Host (as well as any data that
might be a result of the command execution).

2. The controller then de-asserts BSY

3. The Host detects that BSY has been de-asserted and then reads from the controller as many bytes as it wishes (in much the same fashion as it does when writing a command string to the controller: the controller points to the data and the Host moves it).

There is (at least) one implication to this protocol: the Host is capable of tying up 100% of the controller's resources if it so chooses. This is because the controller has no way of knowing when the Host has finished reading/writing from/to it's data buffer. There needs, therefore, to be a mechanism for the Host to let the controller know that it has freed up the controller's resources. This mechanism (for lack of a better name) is called the FreeProcess. The Host communicates the FreeProcess to the controlle in either of two ways: 1) the ProFile way, and 2) the Widget way.

ProFile_FreeProcess ::=

1. The Host downloads a commands of < $F0 > to the controller.

2. The controller decodes the command and enters FreeProcess.

Widget_FreeProcess ::=

1. During the Initial_HandShake (when the controller is attempting to let the Host know that it is ready for a new

command) the Host responds to the $Ø1 with a $69.

2. The controller responds to the reception of a $69 instead of $55 by entering FreeProcess. All further handshaking is terminated.

COMMAND SUMMARY

ProFile_Commands:

```
ProFile_Read ::= ( <$ØØ> < 3 bytes LogicalBlock > )
ProFile_Write ::= ( <$Ø1> < 3 bytes LogicalBlock > )
ProFile_WrVerify ::= ( <$Ø2> < 3 bytes LogicalBlock > )
```

Diagnostic_Commands:

```
Read_ID ::= ( <$12> <$ØØ> <$ED> )
Read_Controller_Status ::= ( <$13> <$Ø1> < Status > < CheckByte > )
Read_Servo_Status ::= ( <$13> <$Ø2> < Status > < CheckByte > )
Send_Servo_Command ::= ( <$16> <$Ø3> < 4 command bytes > < CheckByte > )
Send_Seek ::= ( <$16> <$Ø4> < 4 bytes cyl/head/sector > < CheckByte > )
Send_Restore ::= ( <$13> <$Ø5> < Data/Format Recal > < CheckByte > )
Set_Recovery ::= ( <$13> <$Ø6> < On/Off > < CheckByte > )
Soft_Reset ::= ( <$12> <$Ø7> <$E6> )
Send_Park ::= ( <$12> <$Ø8> <$E5> )
Diag_Read ::= ( <$12> <$Ø9> <$E4> )
Diag_ReadHeader ::= ( <$13> <$ØA> < Sector > < CheckByte > )
Diag_Write ::= ( <$12> <$ØB> <$E2> )
Store_Map ::= ( <$12> <$ØC> < $E1 > )
Read_SpareTable ::= ( <$12> <$ØD> <$EØ> )
Write_SpareTable ::= ( <$16> <$ØE> < PassWord > < CheckByte > )
Format_Track ::= ( <$18> <$ØF>
<Offset><InterLeave><PassWord><CheckByte> )
Initialize_SpareTable ::=
        ( <$16> <$1Ø> < Offset > < InterLeave> < PassWord > <
        CheckByte > )
Read_Abort_Stat ::= ( <$12> <$11> <$DC> )
Reset_Servo ::= ( <$12> <$12> <$DB> )
Scan ::= ( <$12> <$13> <$DA> )
```

System Commands:

```
Sys_Read ::=
        ( <$26> <$ØØ> < BlkCnt > < 3 bytes LogicalBlock > <
        CheckByte > )

Sys_Write ::=
        ( <$26> <$Ø1> < BlkCnt > < 3 bytes LogicalBlock > <
        CheckByte > )

Sys_WrVerify ::= ( <$25> <$Ø2> < 3 bytes LogicalBlock > <
CheckByte > )
```

PassWord ::= < $FØ $78 $3C $1E >

Exception Handling:

Widget has been designed to run fault free for most of it's operating time. This means that almost every single time that a request is made of the controller it will be performed flawlessly. However, there are some exceptional cases - most fall into the category of extreme errors- where the controller must attempt to correct a problem. The most likely to occur is either when the drive is externally 'bumped' and the heads are forced off track, or flaky block is read (crc/ecc error).

SERVO EXCEPTIONS

It is possible for the Servo Processor to detect that the heads have gone off track. When this occurs the Servo will attempt to put the heads back on track transparently to the controller. There are three outcomes to this exception:

1. The Servo will put the heads back on the correct track and all will be well with the world.

2. The Servo will mistakenly put the heads on a track that is close to the target track. In this case the controller will detect a header mismatch the next time it reads a block on the disk and will issue a seek to correct the position error.

3. The Servo will raise ServoError (a gross misalignment detected) and drop ServoReady in which case the controller will have no choice but to issue a DataRecal to clear the ServoError, then issue a seek to get back to the target track.

READ/WRITE EXCEPTIONS

There are occasions when the a spot on the disk surface becomes unuseable, or for some reason causes the data stored in that area to change. To handle this type of exception Widget is equiped with 2 error detecting devices and 1 error correcting device (although Ecc is both error detecting and error correcting). Widget uses a sixteen-bit crc polynomial (CRC-16) to detect all single-burst errors less than sixteen bits in length, almost all single-burst errors of sixteen bits, and most single-burst errors greater than sixteen bits in length. A 48-bit ecc polynomial is also used that has error detecting properties similar to that of the crc polynomial, except that it handles burst of up to 48 bits. It can also correct single-error bursts up to twelve bits in length.

When a block read, if the first read is successful (no errors) then the data is transfered to the Host, thus completing it's command. Suppose, however, that the block is not read successfully the first time. The causes of this exception are 4:

1. Servo Error: this execption is handled by leaving the read routine and getting in touch with the Servo Processor to see if things can be straightened out. Once the controller is convinced that the Servo is well and that the heads are positioned where thye should be, it retries the read.

2. The state machine indicates that it is in the wrong ending state. This is considered a catastrophic exception an the controller will abort.

3. The state machine indicates that a matching header was not found. Before making this decision the state machine searches the track twice for a match header. To handle this exception the controller reads a header from the track that the heads are currently positioned over and tries to determine if the heads are positioned correctly. If they are, then it is assumed that target block's header is faulty and the track will be spared. If no header can be read from the track it can be determined if the heads are positioned correctly or if all headers on the track are shot. In this case the controller will issue a data recal and seek back to the target location and retry. If a header still can not be found the block will be spared.

4. The state machine indicates that a crc or ecc error has occured. The controller will automatically retry 9 times (a total of 10 reads). If a successful read is encountered during this retry session the controller will save the valid data. At the end of all the retries, if the number of bad reads was 2 or less then the block is transfered to the Host. If the number is between 2 and 10 then the data is still returned to the Host,

but the controller goes back to the target block and performs a WriteVerify with the valid data; if the block fails the verify then it is spared. If the number of bad reads is 10 then the ecc correction algorithm is applied to the result of the last retry. If the data is correctable then it is returned to the Host; the target block is then write verified with the valid data and if it fails it is spared. If the data is uncorrectable, then undefined data is returned to the Host (if it chooses to read it) and Standard_Status indicates that the operation failed. The target block is then declared a BadBlock (a form of spare).

BadBlocks have the property that when they are read the controller will attempt to extract the data from the target block and performing exactly the same steps as in a normal read in an attempt to recover the data. When they are written to, the controller performs a write verify to the target block. If the block passes the verifyu then it is no longer a BadBlock, otherwise it is spared.

SpareBlocks have the property that they are 'relocated' logicalblocks. In other words, SpareBlocks are blocks on the disk that are transparent to the Host and were set aside for the explicit purpose of relocating faulty blocks. There are 76 such SpareBlocks on each Widget, spaced 256 blocks apart on a 10MB drive, 512 blocks apart on a 20MB drive, and 1024 blocks apart on the 40MB drive. When I decided upon this sparing algorithm I chose a trade-off between overall performance and data security.

When a block is spared, it is relocated to the nearest available spare block so that the time to get to it is minimized. This works only as long as spared blocks are more or less uniform over the entire disk surface. On the other hand, if the ideal case were to be implemented (the controller keeping track of which blocks on the disk were unused and relocating to the nearest one) the space needed to contain the data structure that kept track of the algporithm would be enormous. The decision to keep the structure contained inside of one data block (512 bytes) led to the 'checker-board' algorithm that has been implemented on Widget.

MISCELLANEOUS

Parking:

To guard against any mishaps when power is shut off to Widget, there is a mechanism in the firmware that takes the heads off the data area of the disk after a period of idleness. This mechanism is known as 'parking'. Unfortunately, it is possible for parking to synchronize with periodic uses of the drive by the Host, causing a mild form of thrashing brought about by the constant seeking needed to move the heads between the park position and the target position. It was determined empirically on ProFile that a good compromise delay time to park is 3 seconds and that time hold for Widget.

Arm_Sweep:

To protect the head-arm bearings from too many short seeks (this causes a possible migration of lubrication away from the surfaces that are meant to be lubricated) the arm is swept the complete width of the disk data surface every 2048 seeks.

Self_Test:

When the controller comes up from being reset it performs the following selftest functions:

1. Register Test
   Write and verify one's and zero'z to all registers; halt if failure
2. Stack Test
   Check push/pop, call/return capabilities; halt if failure
3. Ram Test
   Write ones and zeros to all ram locations; don't allow ProFile or System commands if failure.
4. Eprom Test
   Check external eprom banks 0 and 1 for check byte; don't allow ProFile or System commands if failure.
5. Motor Speed
   Check time from index to index; don't allow ProFile or System commands if failure.
6. Track Count

   Seek from the format recal position to track 0. This test fails if the servo is unable to complete this task.

7. Spare Table

Find both spare tables and write verify them; don't allow ProFile or System commands if failure.

8. Read/Write Test

Widget performs a read/write test on a track not used for data. If a failure occurs on all blocks of that track then the controller assumes that either the disk or the read/write channel is unusable.

APPENDIX C: Abort_Status_Variables

There are occasions when the Widget controller will detect that something is radically wrong with the Widget subsystem, i.e., the ram on-board the controller goes on vacation, or the state machine gives up the ghost, etc. In one of these cases the controller will 'abort' it's current instruction and return control to the Host, hopefully with enough information that the Host can make an intelligent decision concerning the state of the Widget.

The Host can read in some information concerning the abort that the controller took by read Last_Abort_Status. This command returns a result that is 20 bytes long: 4 bytes of Standard_Status followed by 16 bytes of abort status. The contents of the 16 byte result is dependent upon the abort taken, and is determined by examining the contents of the 15th and 16th bytes which are a pointer into the firmware where the abort occured.

In the following list the contents of bytes 15 and 16 are indicated (as a hexadecimal 16-bit integer, just as you would read them from the buffer: byte 15 most significant...), with a brief description of th ereason why the abort was taken as well as any comments concerning other bytes of immediate interest included within the Abort_Status structure.

$02EA: Illegal interface response, or Host Nak
  Byte09: Response Byte received from Host
$03B8: Illegal Ram_Bank select
  Byte00: Bank number of attempted select
$0487: Format Error: Illegal State_Machine State
  Byte0A: State of State_Machine at time of failure
$04CB: Illegal Bank_Switch: Either call or return
  Byte00: Bank number of attempted bank select
$0513: Illegal Interrupt or Dead_Man_Timeout
  Byte0A:0B : Address of routine at time of timeout
$1101: Format Error: Error while writing sector
  Byte09: Error Status from FormatBlock
$11EA: Command CheckByte Error
$1203: ProFile or System command attempted while SelfTest error
$1217: Illegal Interface instruction
$1310: Unrecoverable Servo Error while reading
$13E8: Sparing attempted on non-existent spared block
$1513: Sparing attempted while spare table full
$158D: Deletion attempted of non-existent bad block
$16B4: Illegal exception instruction
$1919: Unrecoverable Servo Error while writing

$1BØ1: Servo Status request sent as Servo Command
$1B56: Restore Error: Non-Recal parameter
    ByteØØ: Illegal parameter sent
$1BAB: Store_Map Error: Parameter larger then the number of sectors
    ByteØA: Illegal parameter sent
$1BD2: Illegal password sent for Write_Spare_Table command
$1C15: Illegal password sent for Format command
$1C24: Illegal format parameters
    Byte09: Offset parameter
    ByteØA: interleave paramter
$1C78: Illegal password sent for Initialize_Spare_Table command
$1CFF: Zero block count sent for MultiBlock transfer
$1E4A: Write Error: Illegal State_Machine state
    ByteØA: State_Machine state at time of error
$1F2F: Read Error: Illegal State_Machine state
    ByteØA: State_Machine state at time of error
$2Ø21: ReadHeader Error: Illegal State_Machine state
    ByteØA: State_Machine state at time of error
$21F7: Request for illegal logical block
    ByteØC: High byte of requested logical block
    ByteØC: Middle byte of requested logical block
    ByteØC: Low byte of requested logical block
$237Ø: Search for SpareTable failed
$2493: No SpareTable structure found in SpareTable
$24B3: UpDate of SpareTable failed
$2522: Illegal SpareCount instruction
    Byte09: Value of illegal instruction
$265E: Unrecoverable Servo Error while performing overlapped seek
$26B8: Unrecoverable Servo Error while seeking
$29EØ: Servo Error after Servo Reset
    ByteØA: Value of controller status port at time of error
$2A1Ø: Servo Communication error after Servo Reset
$2D13: Scan attempted without SpareTable

APPENDIX A

WIDGET SERVO FUNCTIONAL OBJECTIVE

I. BASIC SERVO FUNCTIONS

Widget servo control functions are handled by a Z8 microprocessor. The Z8 handles all I/O operations, timing operations and communication with a host controller. Control functions to the Z8 Servo Controller are made through the serial I/O.

The following commands for the Widget servo are:

A. HOME - not detented, heads off data zones located at the inner stop.

B. RECAL - detented at one of two positions.

   1. FORMAT RECAL: 32, -0, +3 tracks from HOME use only during data formatting.

   2. RECAL: 72, -0, +3 tracks from HOME use to initialize home position after power on or following an access or any other error.

C. SEEK - coarse track positioning of data head to any desired track location.

D. TRACK FOLLOWING - heads are detented on a specific track location and the device is ready for another command.

E. OFFSET - controlled microstepping of fine position system during TRACK FOLLOWING (two modes).

   1. COMMAND OFFSET - direction and amount of offset is specified to the servo.

   2. AUTO OFFSET - command allows the servo to automatically move off track by the amount indicated by the embedded servo signal on the data surface (disk).

F. STATUS - command can read servo status.

G. DIAGNOSTIC - not implemented.

See Table 1 for the actual command description. With the present command structure a SEEK COMMAND can be augmented with an OFFSET COMMAND. Upon completion of a seek, the offset command bit is tested to determine if an offset will occur following a seek (either auto or command offset).

When a SERVO ERROR occurs the Z8 SERVO will attempt to do a short RECAL (ERROR RECAL). Two attempts are made by the system to do the ERROR RECAL function. If either of the two RECAL operations terminate successfully the protocol status will be SERVO READY, SIO READY and SERVO ERROR. Should the ERROR RECAL fail then the system will complete the error recovery by a HOME function.

The two OFFSET commands will be described. First COMMAND OFFSET is a predetermined amount of microstepping of the fine position servo. Included in the OFFSET BYTE (STATREG) bit B6=0 is a COMMAND OFFSET. Bit B7=1 is a forward offset step (toward the spindle); B7=0 is a reverse step. In the case bit B6=1 the OFFSET command is AUTO OFFSET.

AUTO OFFSET command normally occurs during a write operation. When the HDA was initially formated at the factory special encoded servo data was written on each track "near" the index zone. The reason for this follows:

Normal coarse and fine position information for the position servos is derived from an optical signal relative to the actual data head-track location. Over a period of time the relative position (optical signal) will not be aligned to the absolute head-track position by some unknown amount (less than 100 uIn). This small change is important for reliability during the write operation. Write/Read reliability can be degraded due to this misalignment. The special disk encoded servo signal is available to the fine position servo and will correct the difference between the relative position signal of the optics and the absolute head to track position under the data head only at index time. The correction signal can be held indefinitely or updated (if desired at each index time) or until a new OFFSET command or move command (SEEK or RECAL) occurs.

II. COMMUNICATION FUNCTIONS

The servo functions described in the previous section only occur when the servo Z8 microprocessor is in the communication state. Communication states occur immediately after a system reset, upon completing head setting after a recal, seek, offset, read servo status or set servo diagnostic. A special communication state exists after a servo error has occurred. If + SIO READY is not active no communication can exist between the external controller and the servo Z8 processor.

Servo commands are serial bits grouped as five separate bytes total. Refer to Table 1 parts I through V as the total communication string. First byte is the command byte (i.e. seek, read status, recal, etc.). Second byte is the low order difference for a seek (i.e. Byte 2 = $0A is a ten track seek). Third byte is the offset byte (AUTO or COMMAND OFFSET and the magnitude/direction for command offset). Fourth byte is the status and diagnostic byte (use for reading internal servo status or setting diagnostic commands). Byte five is the check sum byte used to check verify that the first four bytes were correctly transmitted (communication error checking).

Part of the communication function requires a specific protocol between the servo Z8 processor and the external controller.

Servo control and communication are described in CHART I. This chart illustrates the basic sequencing and control operations. Chart I does not illustrate the servo error handling or command/protocol handling functions. Error handling is described in Section IV and illustrated by CHART II.

III. Z8 SERVO PROTOCOL

The protocol between the Z8 SERVO microcomputer and the CONTROLLER is based on five I/O lines. Two of the I/O lines are serial input (to Z8 servo from controller) serial output (from Z8 servo to controller). Data stream between the Z8 servo and controller is 8 bit ACSII with no parity bit (the fifth byte of the command string contains check sum byte use for error checking). There are three additional output lines between the Z8 servo used as control lines to the controller. Combining the two serial I/O lines and the three unidirectional port lines generates the bases of the protocol between the Z8 servo and controller. The important operations between the Z8 servo and controller are:

1. Send commands to Z8 servo.

2. Read Z8 servo status.

3. Check validity of all four command bytes.

4. I/O timing signals between the Z8 servo and controller.

5. Z8 servo reset.

Sequencing the Z8 servo controller is an important process following a Power Up (Power On Reset) or if the controller should issue a Z8 Servo Reset at any time. After a Z8 Servo Reset is inhibited the Z8 I/O ports and internal register are initialized. This takes approximately 75 msec after the Z8 Servo Reset is inhibited. The protocol baud rate is automatically set to 19.2KB and then the system is parked at HOME position and SIO READY is set active. ***IMPORTANT***. If the desired baud rate needs to be increased to 57.6KB; **after a Z8 Servo Reset is the ONLY time this can be done***. Once set to 57.6KB tne communication rate remains at 57.6KB until a Z8 Servo Reset occurs. Setting 57.6KB is achieved as follows:

1. Z8 Servo "Power On or Controller" Reset

2. Wait for SIO Ready

3. Send a READ STATUS COMMAND as follows:

   BYTE 1 = $ 00
   BYTE 2 = $ 00
   BYTE 3 = $ 00
   BYTE 4 = $ 87

After the completion of transmitting the bytes, the Z8 Servo Controller chanzges to 57.6KB and will be waiting for the next transmitted command at 57.6KB.

Before the controller transmits the command byte the controller must pole the SIO READY line from the Z8 servo to determine if it is active (+5 volts). If the line is active then a command can be transmitted to the Z8 servo. The program in the Z8 servo will determine what to do with the command bytes (depending upon the current status of the Z8 servo). After the command (five bytes long) has been transmitted to the Z8 servo, the program in the Z8 servo will determine if the command bytes (first four bytes) are in error by evaluating the check sum byte (fifth byte transmitted). See table Chart III and IV for the error handling. After the controller has transmitted the last serial string it must wait 250 usec then test for SERVO ERROR active (+5 volts). If SERVO ERROR is active the command was rejected (check sum error or invalid command). If the SERVO ERROR is set active 600μsec after the command is sent (and not 250 sec), this was a command reject. The SERVO ERROR must be cleared by READ STATUS COMMAND or RECAL COMMAND before transmitting another command. See CHART 1 for time diagram of the command sequence and I/O protocol.

As long as SIO READY is active the controller can communicate with the Z8 Servo Controller. If SERVO READY is not active the only command that will cause the Widget Servo to set SERVO READY active is a RECAL COMMAND (NORMAL or FORMAT). Read Status will only clear SERVO ERROR. And all other commands will be rejected.

Next, if SERVO READY is active and SERVO ERROR is also active, SERVO ERROR can be cleared by:

1. Any READ STATUS COMMAND.

2. Any RECAL COMMAND.

3. Any other commands will be rejected and maintain SERVO ERROR.

If a SEEK COMMAND is transmitted with both SERVO READY and SERVO ERROR active the command will be rejected.

It is important to check the status of all three status lines from the Z8 Servo. It is best to avoid sending a SEEK COMMAND with SERVO READY and SERVO ERROR active.

Chart V parts A-I illustrate some of the serial communication commands and error conditions that can occur between the controller and Z8 SERVO.

IV. ERROR HANDLING

SERVO ERROR will be generated during the following conditions:

1. During Recal mode (velocity control only) access time-out.If a Recal function exceeds 150 msec then an access timeout occurs.

2. During Seek mode (velocity control only) access time-out. If a Seek function exceeds 150 msec then an access time-out occurs.

3. During Settling mode (following a Recal, Seek, or Offset) if there is excessive On Track pulses (3 crossings) indicating excessive head motion a Settling error check will occur.

4. During a command transmission if a communication error occurs (check sum error).

5. During a command tansmission if a invalid command is sent.

APPENDIX A:

I. The purpose of the FINE POSITION SERVO is to maintain detent or lock on a given data track. Any misregistrations of the head/arm due to windage, mechanical observed by the optics position signal are corrected by the close loop position servo. Misregistrations at the data head relative to the actual data track on the disk must be corrected by the AUTO OFFSET command. Figure I illustrates a block diagram of the Widget FINE POSITION SERVO. The amount of misregistration at the data track sensed after a AUTO OFFSET command are summed into the servo and the servo is automatically repositioned over the data track.

II. The COARSE POSITION SERVO (SEEK) has the function of moving the data head arbitarily from a current track to any other arbitrary track location within the total number of track locations between the inner to outer crash stops. When a command is transmitted to the Z8 Servo controller, the Z8 decodes and interprets the command into a servo function. If a SEEK command is sent to the Z8 Servo Controller a direction and number of tracks to move is also sent. The system starts its move to the new track location. When the arm has moved to its new location the Z8 Servo Controller provides control and delay necessary to allow the data head and the FINE POSITION SERVO to come to rest immediately following a SEEK. This insures that motion in FINE POSITION SERVO and data head will be under control when the READ/WRITE channel begins operation. Reliability of the data channel is assured with high margins. Figure I illustrates a block diagram of the Widget COARSE POSITION SERVO.

The differences between the FINE POSITION SERVO and the COARSE POSITION SERVO is handled by the Z8 Servo Controller. The two servos share for the most part the same set of electronics. The Z8 Servo Controller and analog multiplexers switch between the signal paths. In general there are some circuits that are not shared because of their uniqueness for a particular servo.

I. BYTE 1: COMMAND BYTE (DIFCNTH)

```
                                  ---
          ---                    | B7  B6  B5  B4 |  FUNCTIONS
         |B7                     |---------------------------------------
         |B6                     | 1   0   0   0  | access only
command  |B5                     | 1   0   0   1  | access with offset
bits     |B4                     | 0   1   0   0  | normal recal (to trk 72)
          ---                    | 0   1   1   1  | format recal (to trk 32)
          ---                    | 0   0   0   1  | offset-trk following
         |B3 -X- not used         | 1   1   0   0  | home-send to ID stop
access   |B2 -access direction    | 0   0   1   0  | diagnostic command
bits     |B1 -hi diff2 (512)      | 0   0   0   0  | read status command
         |B0 -hi diff1 (256)      ---------------------------------------
          ---
```

access direction = 1 (FORWARD: toward the spindle)
= 0 (REVERSE: away from the spindle)

hi diff2 (512) = 1 (512 tracks to go)
= 0 (not set)

hi diff1 (256) = 1 (256 tracks to go)
= 0 (not set)

II. BYTE 2: DIFF BYTE (DIFCNTL)

command BYTE 2 contains the LOW ORDER DIFFERENCE COUNT for a seek

```
 ---
|B7 -bit7= 128 tracks
|B6 -bit6= 64  tracks
|B5 -bit5= 32  tracks
|B4 -bit4= 16  tracks
|B3 -bit3= 8   tracks
|B2 -bit2= 4   tracks
|B1 -bit1= 2   tracks
|B0 -bit0= 1   track
 ---
```

WIDGET SERVO FUNCTIONAL OBJECTIVE

I. BASIC SERVO FUNCTIONS

Widget servo control functions are handled by a Z8 microprocessor. The Z8 handles all I/O operations, timing operations and communication with a host controller. Control functions to the Z8 Servo Controller are made through the serial I/O.

The following commands for the Widget servo are:

A. HOME - not detented, heads off data zones located at the inner stop.

B. RECAL - detented at one of two positions.

   1. FORMAT RECAL: 32, -0, +3 tracks from HOME use only during data formatting.

   2. RECAL: 72, -0, +3 tracks from HOME use to initialize home position after power on or following an access or any other error.

C. SEEK - coarse track positioning of data head to any desired track location.

D. TRACK FOLLOWING - heads are detented on a specific track location and the device is ready for another command.

E. OFFSET - controlled microstepping of fine position system during TRACK FOLLOWING (two modes).

   1. COMMAND OFFSET - direction and amount of offset is specified to the servo.

   2. AUTO OFFSET - command allows the servo to automatically move off track by the amount indicated by the embedded servo signal on the data surface (disk).

F. STATUS - command can read servo status.

G. DIAGNOSTIC - not implemented.

See Table 1 for the actual command description. With the present command structure a SEEK COMMAND can be augmented with an OFFSET COMMAND. Upon completion of a seek, the offset command bit is tested to determine if an offset will occur following a seek (either auto or command offset).

When a SERVO ERROR occurs the Z8 SERVO will attempt to do a short RECAL (ERROR RECAL). Two attempts are made by the system to do the ERROR RECAL function. If either of the two RECAL operations terminate successfully the protocol status will be SERVO READY, SIO READY and SERVO ERROR. Should the ERROR RECAL fail then the system will complete the error recovery by a HOME function.

The two OFFSET commands will be described. First COMMAND OFFSET is a predetermined amount of microstepping of the fine position servo. Included in the OFFSET BYTE (STATREG) bit B6=0 is a COMMAND OFFSET. Bit B7=1 is a forward offset step (toward the spindle); B7=0 is a reverse step. In the case bit B6=1 the OFFSET command is AUTO OFFSET.

AUTO OFFSET command normally occurs during a write operation. When the HDA was initially formated at the factory special encoded servo data was written on each track "near" the index zone. The reason for this follows:

Normal coarse and fine position information for the position servos is derived from an optical signal relative to the actual data head-track location. Over a period of time the relative position (optical signal) will not be aligned to the absolute head-track position by some unknown amount (less than 100 uIn). This small change is important for reliability during the write operation. Write/Read reliability can be degraded due to this misalignment. The special disk encoded servo signal is available to the fine position servo and will correct the difference between the relative position signal of the optics and the absolute head to track position under the data head only at index time. The correction signal can be held indefinitely or updated (if desired at each index time) or until a new OFFSET command or move command (SEEK or RECAL) occurs.

II. COMMUNICATION FUNCTIONS

The servo functions described in the previous section only occur when the servo Z8 microprocessor is in the communication state. Communication states occur immediately after a system reset, upon completing head setting after a recal, seek, offset, read servo status or set servo diagnostic. A special communication state exists after a servo error has occurred. If + SIO READY is not active no communication can exist between the external controller and the servo Z8 processor.

Servo commands are serial bits grouped as five separate bytes total. Refer to Table 1 parts I through V as the total communication string. First byte is the command byte (i.e. seek, read status, recal, etc.). Second byte is the low order difference for a seek (i.e. Byte 2 = $0A is a ten track seek). Third byte is the offset byte (AUTO or COMMAND OFFSET and the magnitude/direction for command offset). Fourth byte is the status and diagnostic byte (use for reading internal servo status or setting diagnostic commands). Byte five is the check sum byte used to check verify that the first four bytes were correctly transmitted (communication error checking).

Part of the communication function requires a specific protocol between the servo Z8 processor and the external controller.

Servo control and communication are described in CHART I. This chart illustrates the basic sequencing and control operations. Chart I does not illustrate the servo error handling or command/protocol handling functions. Error handling is described in Section IV and illustrated by CHART II.

III. Z8 SERVO PROTOCOL

The protocol between the Z8 SERVO microcomputer and the CONTROLLER is based on five I/O lines. Two of the I/O lines are serial input (to Z8 servo from controller) serial output (from Z8 servo to controller). Data stream between the Z8 servo and controller is 8 bit ACSII with no parity bit (the fifth byte of the command string contains check sum byte use for error checking). There are three additional output lines between the Z8 servo used as control lines to the controller. Combining the two serial I/O lines and the three unidirectional port lines generates the bases of the protocol between the Z8 servo and controller. The important operations between the Z8 servo and controller are:

1. Send commands to Z8 servo.

2. Read Z8 servo status.

3. Check validity of all four command bytes.

4. I/O timing signals between the Z8 servo and controller.

5. Z8 servo reset.

Sequencing the Z8 servo controller is an important process following a Power Up (Power On Reset) or if the controller should issue a Z8 Servo Reset at any time. After a Z8 Servo Reset is inhibited the Z8 I/O ports and internal register are initialized. This takes approximately 75 msec after the Z8 Servo Reset is inhibited. The protocol baud rate is automatically set to 19.2KB and then the system is parked at HOME position and SIO READY is set active. ***IMPORTANT***. If the desired baud rate needs to be increased to 57.6KB; **after a Z8 Servo Reset is the ONLY time this can be done***. Once set to 57.6KB the communication rate remains at 57.6KB until a Z8 Servo Reset occurs. Setting 57.6KB is achieved as follows:

1. Z8 Servo "Power On or Controller" Reset

2. Wait for SIO Ready

3. Send a READ STATUS COMMAND as follows:

   BYTE 1 = $ 00
   BYTE 2 = $ 00
   BYTE 3 = $ 00
   BYTE 4 = $ 87

After the completion of transmitting the bytes, the Z8 Servo Controller chanzges to 57.6KB and will be waiting for the next transmitted command at 57.6KB.

Before the controller transmits the command byte the controller must pole the SIO READY line from the Z8 servo to determine if it is active (+5 volts). If the line is active then a command can be transmitted to the Z8 servo. The program in the Z8 servo will determine what to do with the command bytes (depending upon the current status of the Z8 servo). After the command (five bytes long) has been transmitted to the Z8 servo, the program in the Z8 servo will determine if the command bytes (first four bytes) are in error by evaluating the check sum byte (fifth byte transmitted). See table Chart III and IV for the error handling. After the controller has transmitted the last serial string it must wait 250 usec then test for SERVO ERROR active (+5 volts). If SERVO ERROR is active the command was rejected (check sum error or invalid command). If the SERVO ERROR is set active 600μsec after the command is sent (and not 250 sec), this was a command reject. The SERVO ERROR must be cleared by READ STATUS COMMAND or RECAL COMMAND before transmitting another command. See CHART 1 for time diagram of the command sequence and I/O protocol.

As long as SIO READY is active the controller can communicate with the Z8 Servo Controller. If SERVO READY is not active the only command that will cause the Widget Servo to set SERVO READY active is a RECAL COMMAND (NORMAL or FORMAT). Read Status will only clear SERVO ERROR. And all other commands will be rejected.

Next, if SERVO READY is active and SERVO ERROR is also active, SERVO ERROR can be cleared by:

1. Any READ STATUS COMMAND.

2. Any RECAL COMMAND.

3. Any other commands will be rejected and maintain SERVO ERROR.

If a SEEK COMMAND is transmitted with both SERVO READY and SERVO ERROR active the command will be rejected.

It is important to check the status of all three status lines from the Z8 Servo. It is best to avoid sending a SEEK COMMAND with SERVO READY and SERVO ERROR active.

Chart V parts A-I illustrate some of the serial communication commands and error conditions that can occur between the controller and Z8 SERVO.

IV. ERROR HANDLING

SERVO ERROR will be generated during the following conditions:

1. During Recal mode (velocity control only) access time-out.If a Recal function exceeds 150 msec then an access timeout occurs.

2. During Seek mode (velocity control only) access time-out. If a Seek function exceeds 150 msec then an access time-out occurs.

3. During Settling mode (following a Recal, Seek, or Offset) if there is excessive On Track pulses (3 crossings) indicating excessive head motion a Settling error check will occur.

4. During a command transmission if a communication error occurs (check sum error).

5. During a command tansmission if a invalid command is sent.

APPENDIX A:

I. The purpose of the FINE POSITION SERVO is to maintain detent or lock on a given data track. Any misregistrations of the head/arm due to windage, mechanical observed by the optics position signal are corrected by the close loop position servo. Misregistrations at the data head relative to the actual data track on the disk must be corrected by the AUTO OFFSET command. Figure I illustrates a block diagram of the Widget FINE POSITION SERVO. The amount of misregistration at the data track sensed after a AUTO OFFSET command are summed into the servo and the servo is automatically repositioned over the data track.

II. The COARSE POSITION SERVO (SEEK) has the function of moving the data head arbitarily from a current track to any other arbitrary track location within the total number of track locations between the inner to outer crash stops. When a command is transmitted to the Z8 Servo controller, the Z8 decodes and interprets the command into a servo function. If a SEEK command is sent to the Z8 Servo Controller a direction and number of tracks to move is also sent. The system starts its move to the new track location. When the arm has moved to its new location the Z8 Servo Controller provides control and delay necessary to allow the data head and the FINE POSITION SERVO to come to rest immediately following a SEEK. This insures that motion in FINE POSITION SERVO and data head will be under control when the READ/WRITE channel begins operation. Reliability of the data channel is assured with high margins. Figure I illustrates a block diagram of the Widget COARSE POSITION SERVO.

The differences between the FINE POSITION SERVO and the COARSE POSITION SERVO is handled by the Z8 Servo Controller. The two servos share for the most part the same set of electronics. The Z8 Servo Controller and analog multiplexers switch between the signal paths. In general there are some circuits that are not shared because of their uniqueness for a particular servo.

APPENDIX B:

An important part of the Widget Servo System is the optics signal. The optics signal provides the necessary signals for the five position servo position the data head accurately over the data track and to provide the system velocity signal during seek mode. The alignment of the optics signal is described in the following section on "WIDGET OPTICS ALIGNMENT PROCEDURE."

WIDGET SERVO

VARIOUS KEY WAVEFORMS

CONTENTS

WAVEFORM: Optics Adjustment

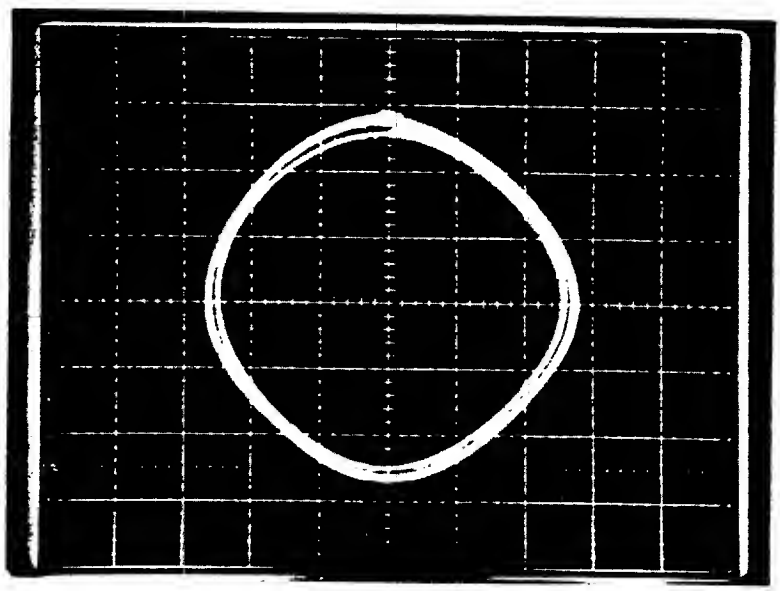
Scope Adjustments:

Channel	Probe Tip	Test Point	Notes
Chan 1	Position A	TP9	2V/div
Chan 2	Position B	TP8	2V/div
Trig In	Not used		
Horiz :	X-Y Mode		

Servo:

Alternate Seeks, 512 tracks

Press Z; 82, 0, 0, 0
86, 0, 0, 0



WAVEFORM: Current Sense and Position A

Scope Adjustments:

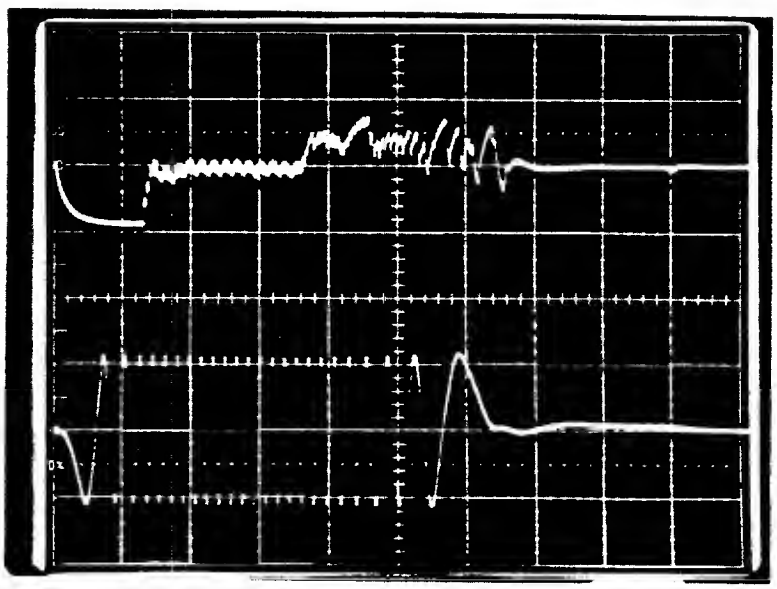
Channel	Probe Tip	Test Point	Notes
Chan 1	Current Sense	TP19	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM1: Current Sense and Position A
(Forward and Reverse Seeks)

Scope Adjustments:

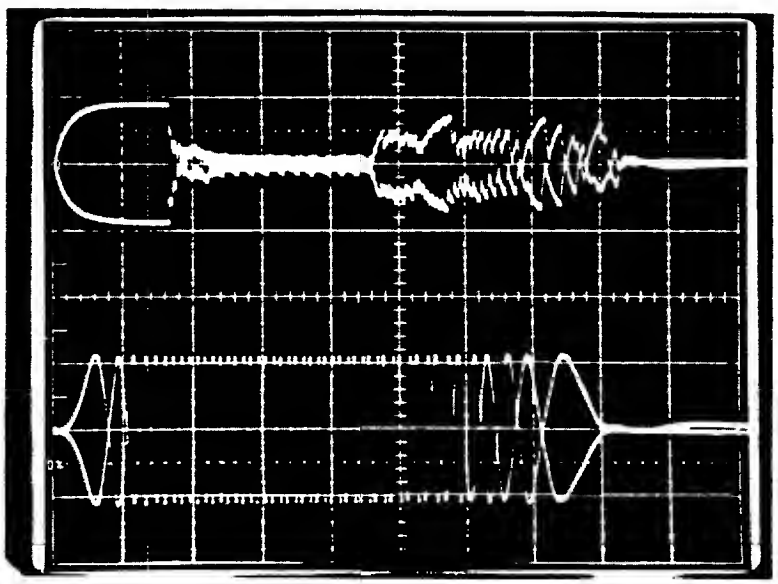
Channel	Probe Tip	Test Point	Notes
Chan 1	Current Sense	TP19	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Velocity and Position A

Scope Adjustments:

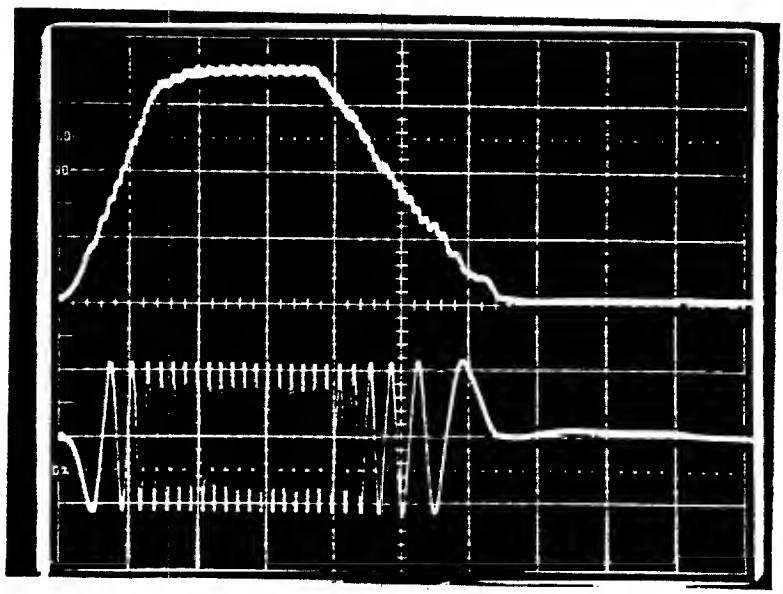
Channel	Probe Tip	Test Point	Notes
Chan 1	Velocity	TP7	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM1: Velocity and Position A
(Forward and Rev Seeks)

Scope Adjustments:

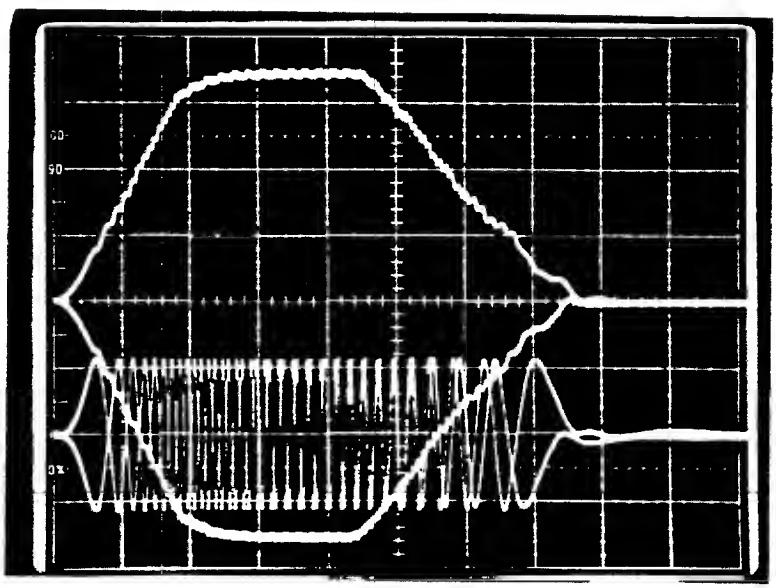
Channel	Probe Tip	Test Point	Notes
Chan 1	Velocity	TP7	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: DAC Output and Position A

Scope Adjustments:

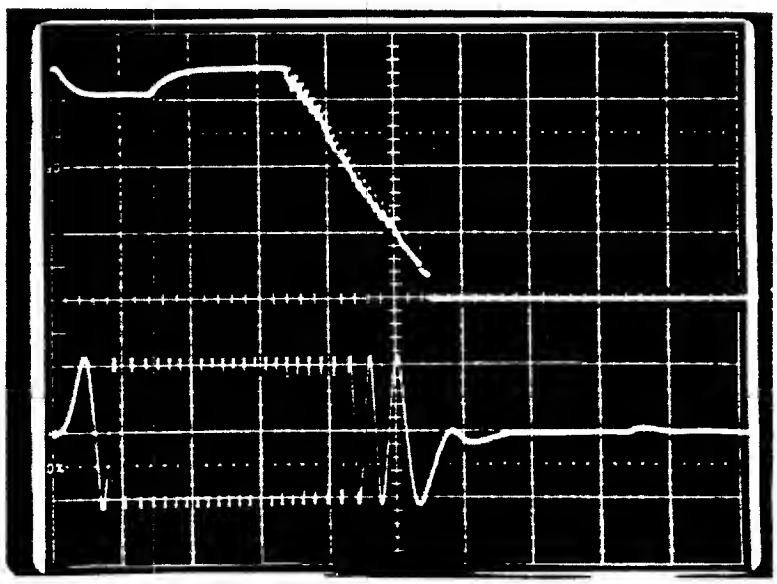
Channel	Probe Tip	Test Point	Notes
Chan 1	DAC Output	TP13	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: DAC Output and Position A
(Forward and Rev Seeks)

Scope Adjustments:

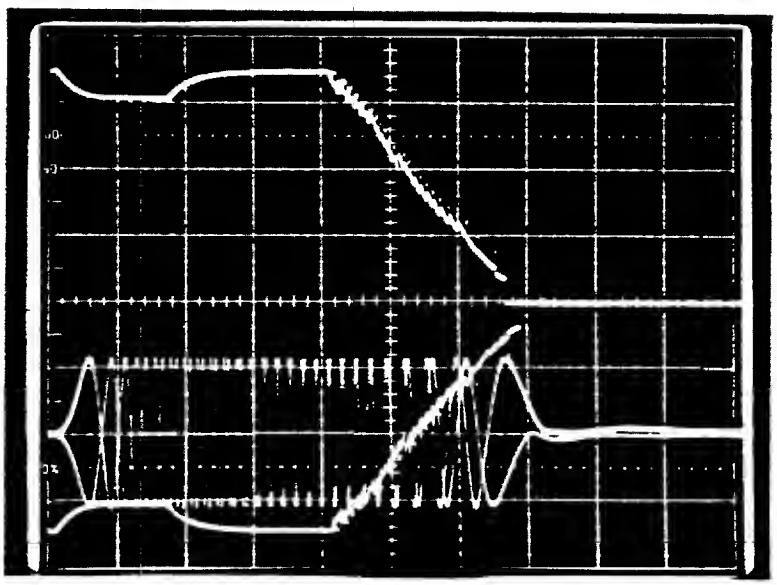
Channel	Probe Tip	Test Point	Notes
Chan 1	DAC Output	TP13	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Curve Shift Function and Position A
(Forward and Rev Seeks: 1 track)

Scope Adjustments:

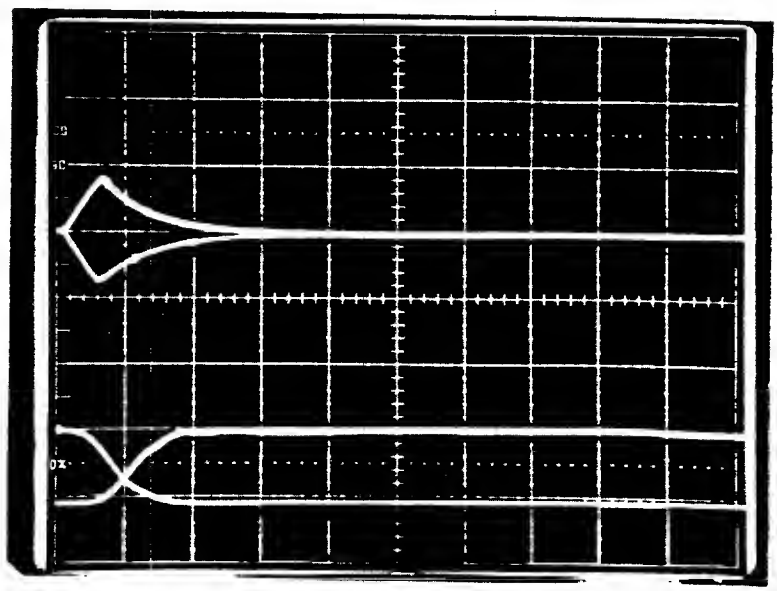
Channel	Probe Tip	Test Point	Notes
Chan 1	Curve Shift Func.	TP12	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 1 track

Press Z; 80, 01, 0, 0
84, 01, 0, 0



WAVEFORM: Curve Shift Function and Position A
(60 track seek)

Scope Adjustments:

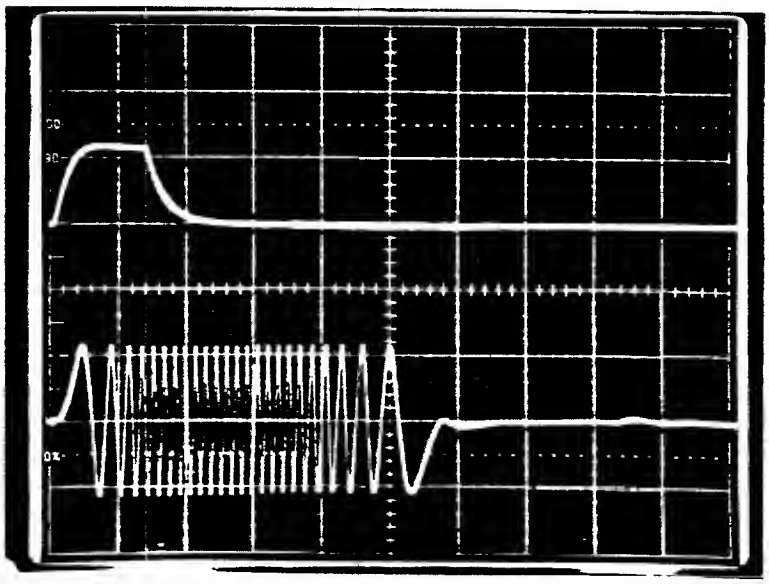
Channel	Probe Tip	Test Point	Notes
Chan 1	Curve Shift Func.	TP12	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



I. BYTE 1: COMMAND BYTE (DIFCNTH)

```
           ---
          |B7
command   |B6
bits      |B5
          |B4
           ---
           ---
          |B3 -X- not used
access    |B2 -access direction
bits      |B1 -hi diff2 (512)
          |B0 -hi diff1 (256)
           ---
```

	B7	B6	B5	B4	FUNCTIONS
8	1	0	0	0	access only
9	1	0	0	1	access with offset
4	0	1	0	0	normal recal (to trk 72)
7	0	1	1	1	format recal (to trk 32)
1	0	0	0	1	offset-trk following
C	1	1	0	0	home-send to ID stop
2	0	0	1	0	diagnostic command
0	0	0	0	0	read status command

```
access direction = 1 (FORWARD: toward the spindle)
                 = 0 (REVERSE: away from the spindle)

hi diff2 (512)   = 1 (512 tracks to go)
                 = 0 (not set)

hi diff1 (256)   = 1 (256 tracks to go)
                 = 0 (not set)
```

II. BYTE 2: DIFF BYTE (DIFCNTL)

command BYTE 2 contains the LOW ORDER DIFFERENCE COUNT for a seek

```
 ---
|B7 -bit7= 128 tracks
|B6 -bit6= 64  tracks
|B5 -bit5= 32  tracks
|B4 -bit4= 16  tracks
|B3 -bit3= 8   tracks
|B2 -bit2= 4   tracks
|B1 -bit1= 2   tracks
|B0 -bit0= 1   track
 ---
```

III. BYTE 3: OFFSET BYTE (STATREG)

command BYTE 3 contains the INSTRUCTION for an OFFSET COMMAND (seek or during track following)

```
 ---
|B7 -offset direction
|B6 -auto offset function
|B5 -read offset value (after auto or manual)
|B4 -offset bit4 =16
|B3 -offset bit3 =8
|B2 -offset bit2 =4
|B1 -offset bit1 =2
|B0 -offset bit0 =1
 ---
```

1. if offset command from BYTE 1 is followed by bit6 set (auto offset) offset direction (bit7) read offset (bit5) and bits 4-0 are ignored but should be set to 0 if not used.

2. OFFSET DIRECTION =1 (FORWARD OFFSET:toward the spindle)
   =0 (REVERSE OFFSET:away from the spindle)

3. AUTO OFFSET =1 (normally used preceeding a write operation)
   =0 (manual offset:MUST send direction and magnitud of offset)

4. READ OFFSET =1 (read offset value from DAC;i.e. after auto offset)
   =0 (no action)

* READ OFFSET COMMAND desired after AUTO OFFSET MUST be sent as two seperate commands

IV. BYTE 4: STATUS BYTE (CNTREG)

```
 ---
|B7 -communication rate
|B6 -power on reset
|B5 -not used
|B4 -not used
|B3 -status or diagnostic bits
|B2 -          |
|B1 -          |
|B0 -          V
 ---
```

B7=0; Communication Rate is 19.2 KBAUD
  =1; Communication Rate is 57.6 KBAUD

B6=0; Power On Reset bit is no active
  =1; Power On Reset bit is active

V. BYTE 5: CHECKSUM BYTE (CKSUM)

(B7 B6 B5 B4 B3 B2 B1 B0)

results of the transmitted CHECKSUM BYTE are derived as:

$$\overline{(\text{BYTE 1} + \text{BYTE 2} + \text{BYTE 3} + \text{BYTE 4})} = \text{CHECKSUM BYTE}$$

(+) is defined as the addition of each BYTE

$\overline{(\text{BYTE})}$ is defined as the compliment of the BYTE (1-4)

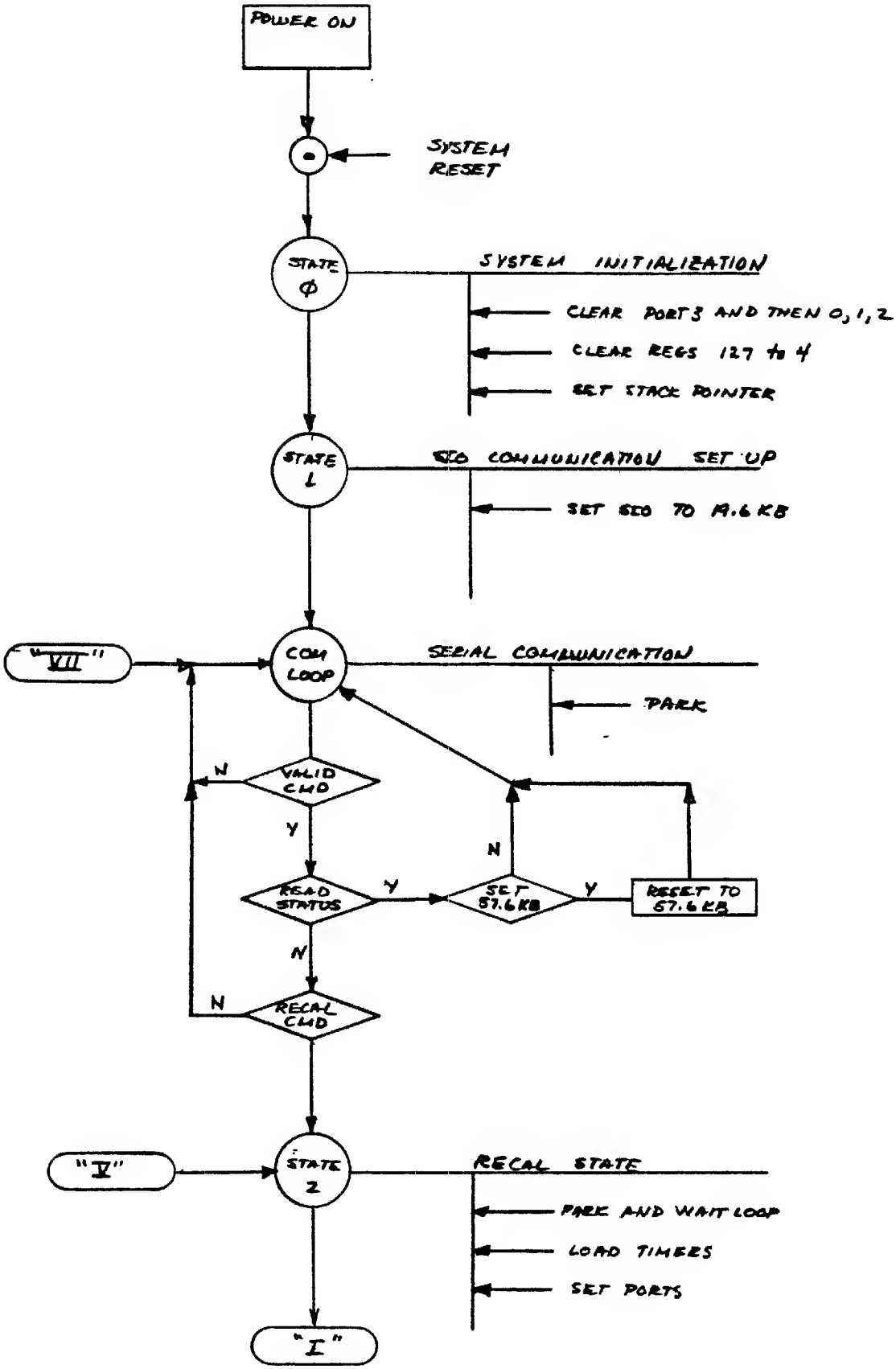
VI. The SERVO STATUS lines (SIO RDY,SERVO RDY,SERVO ERROR) must have the following conditions in order to send the listed Z8 COMMANDS:

Z8 SERVO CMD	HEX	SERVO STATUS SIO RDY	SERVO RDY	SERVO ERROR	
access(only)	8X	1	1	0	
access(offset)	9X	1	1	0	
recal(data)	40	1	X	X	
recal(format)	70	1	X	X	
park	C0	1	X	X	
offset(detent)	10	1	1	0	
status	00	1	X	X	
diagnostic	20				not implimented

X= either 0,1

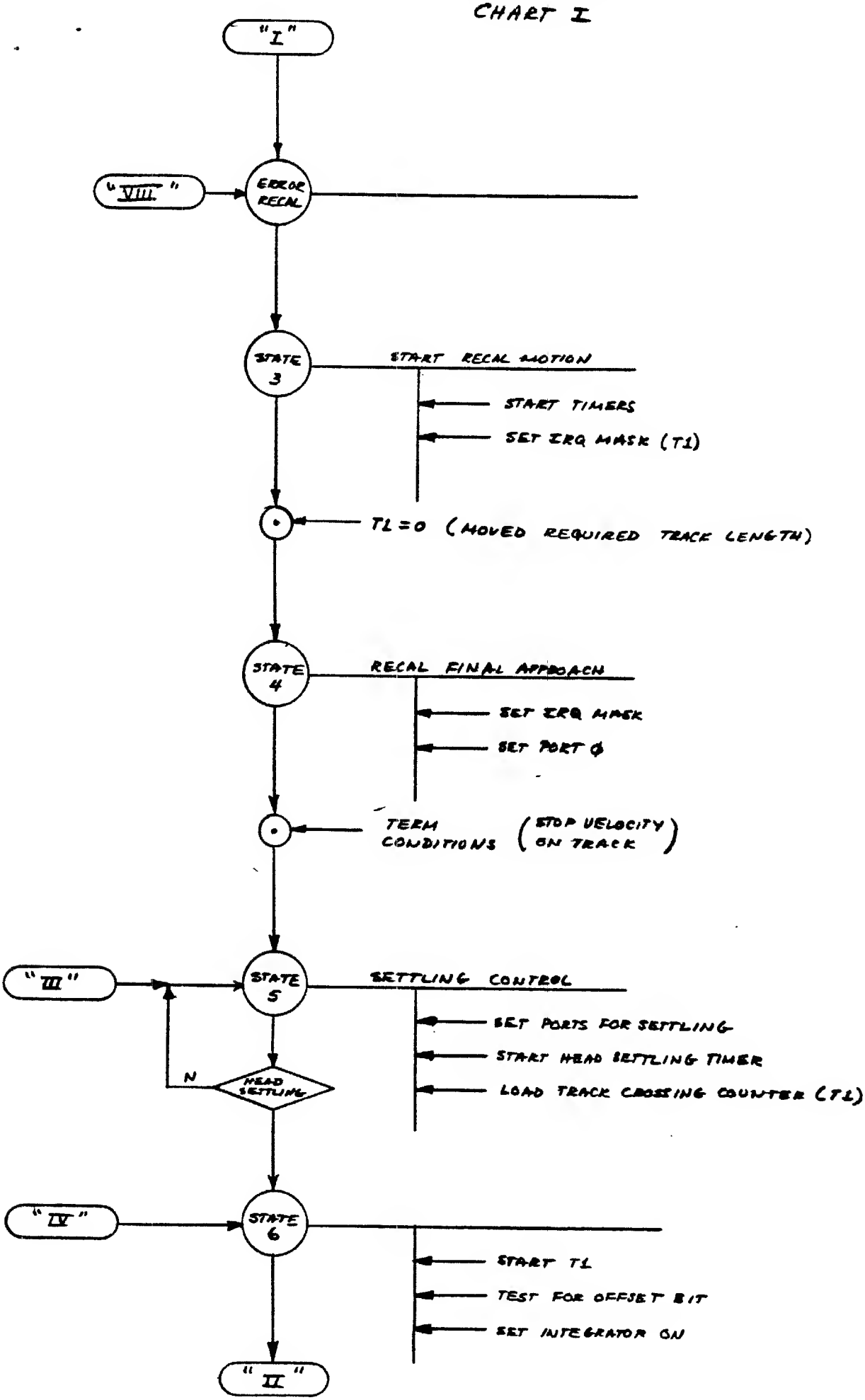
8 SERVO SEQUENCER

CHART I



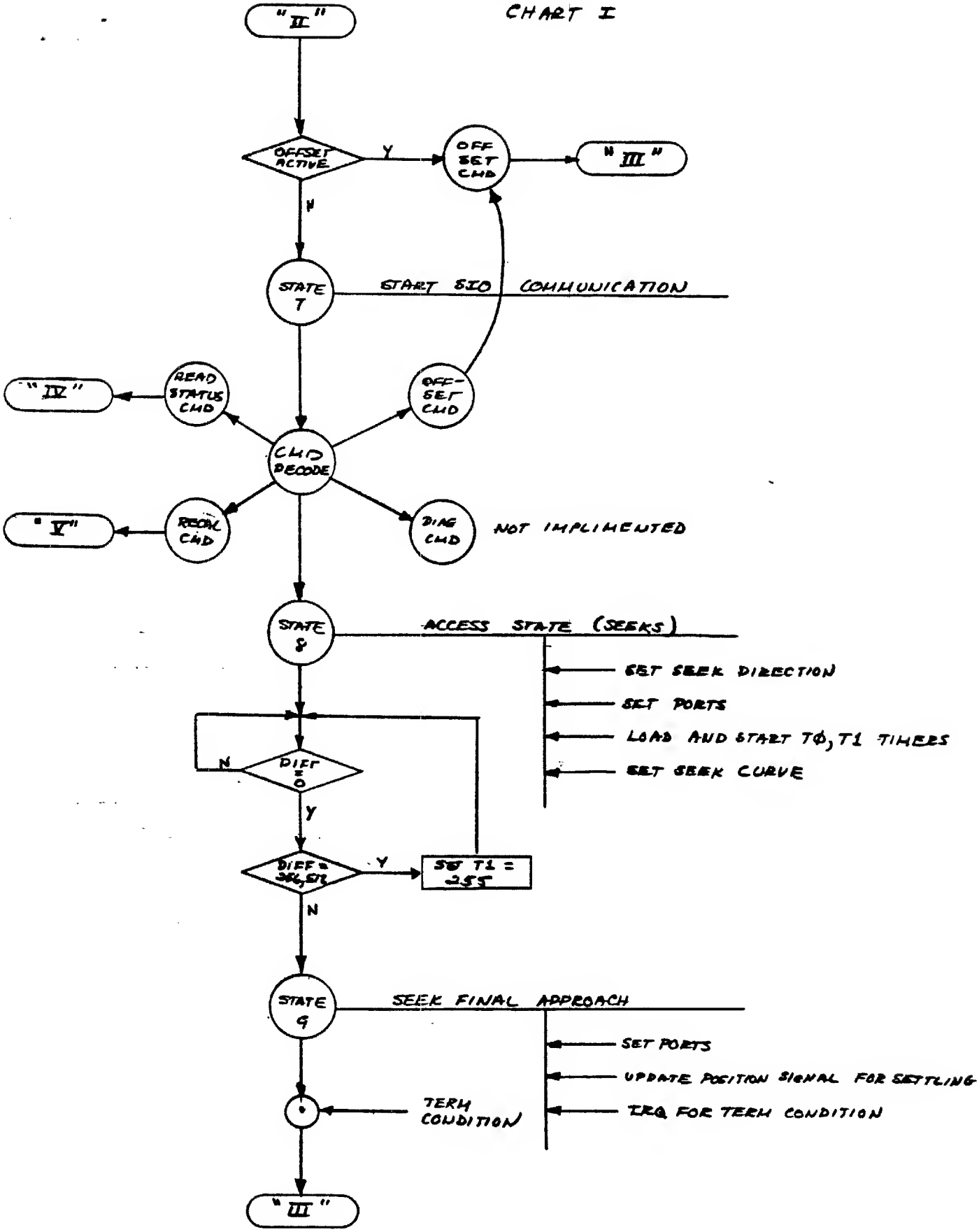
E8 SERVO SEQUENCER

CHART I

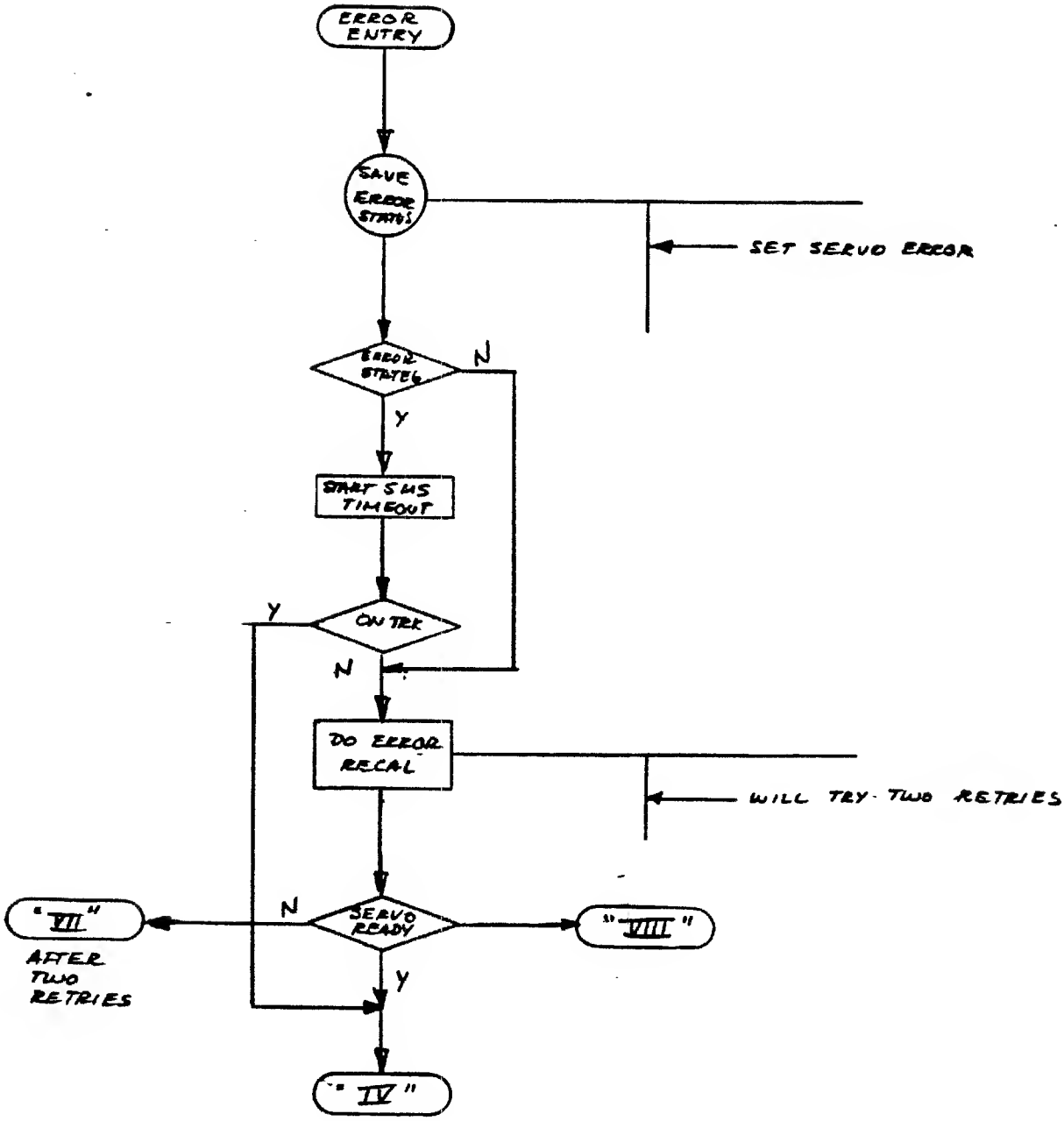


Z8 SERVO SEQUENCER

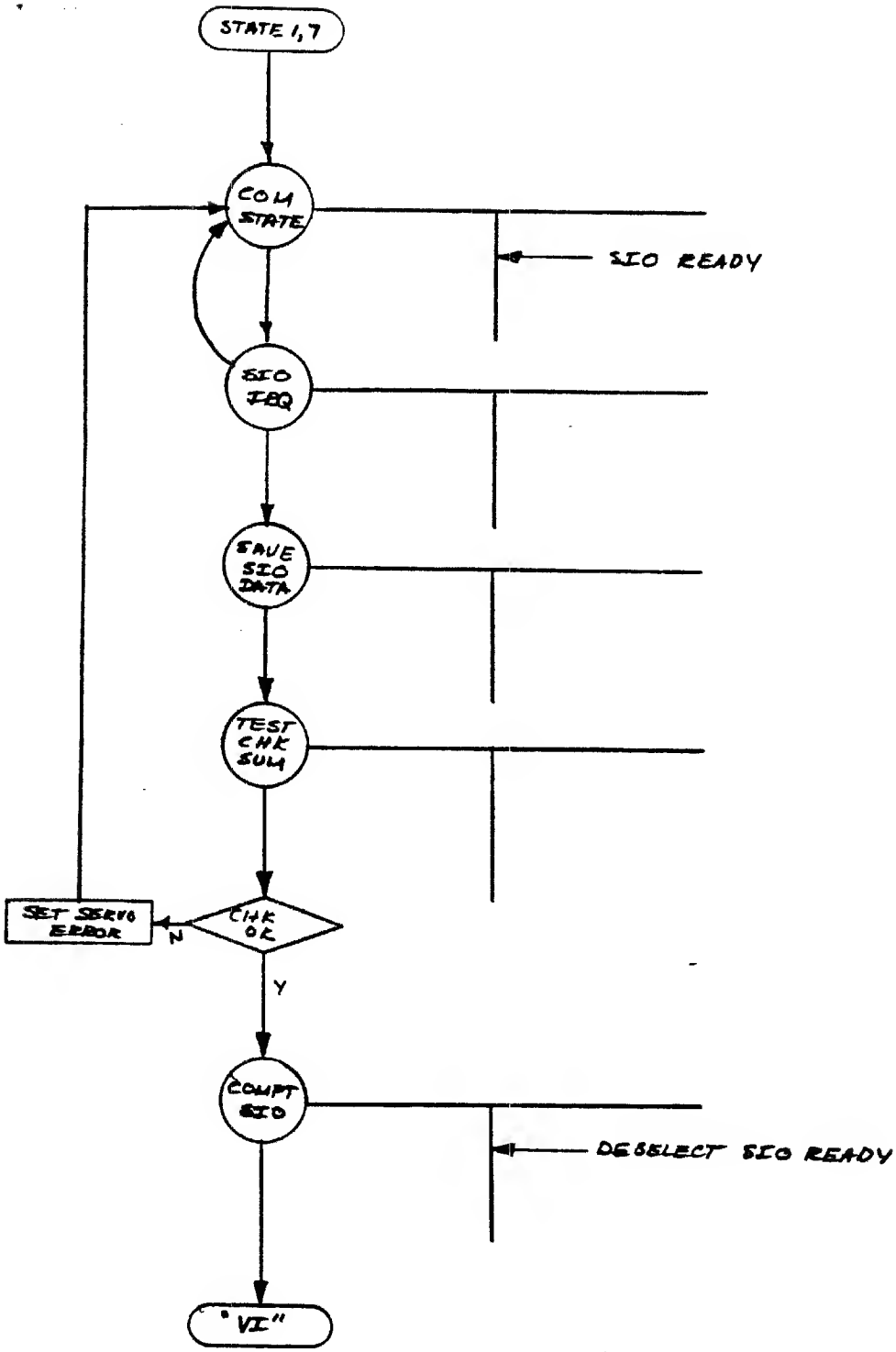
CHART I



SERVO ERROR
CHART II



COMMUNICATION ERRORS
CHART III



COMMAND ERRORS

CHART IV

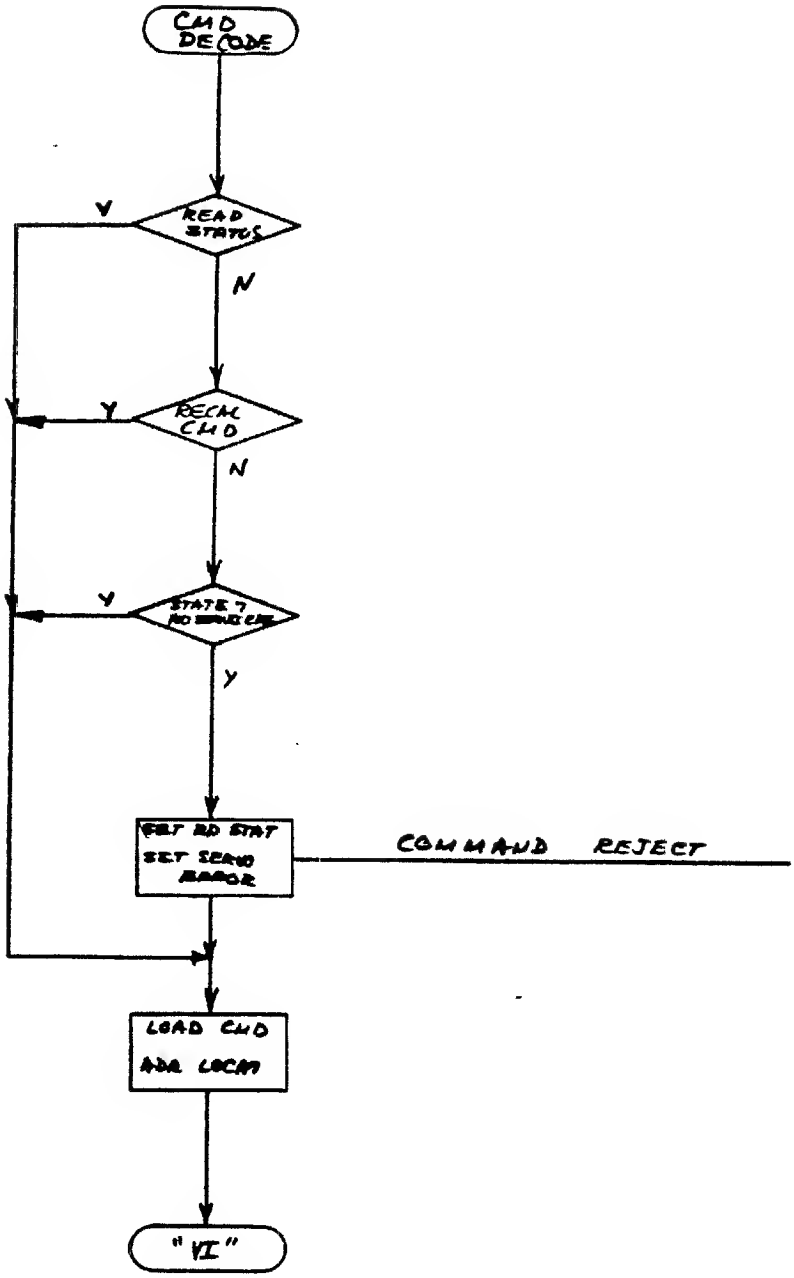
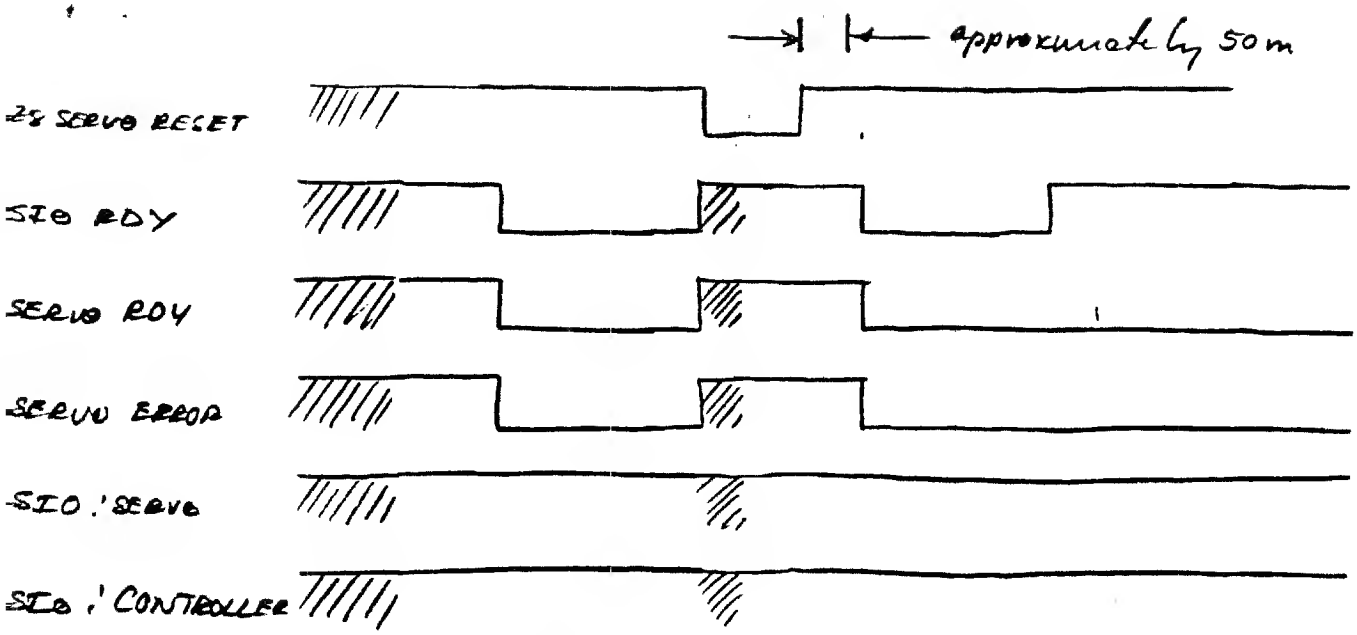
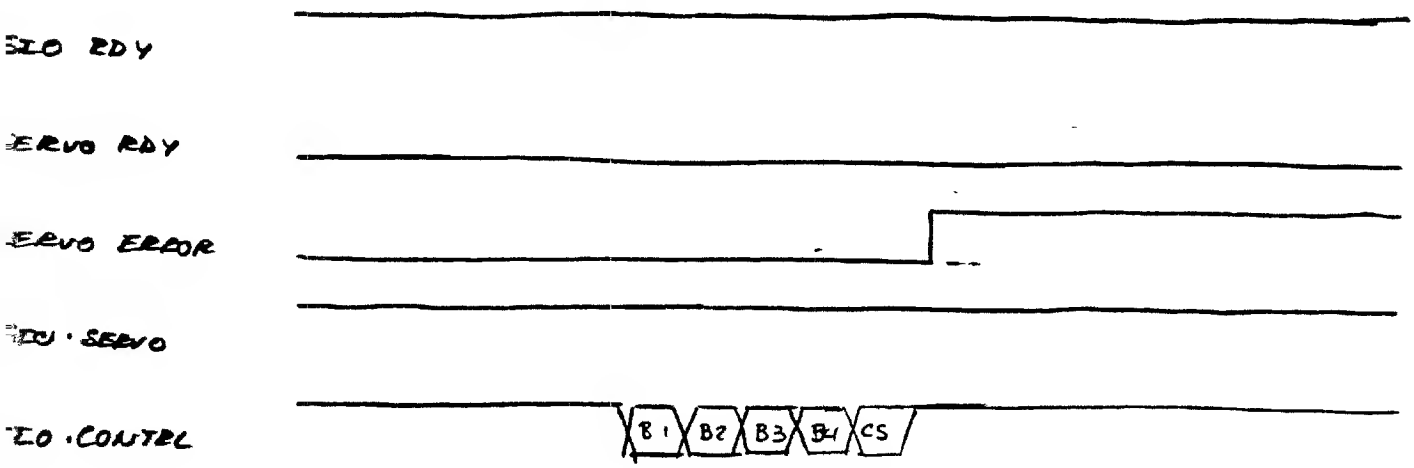


CHART V

A - POWER UP



B - AFTER POWER UP - CHECK SIO ERROR



C - AFTER POWER UP - INVALID CMD

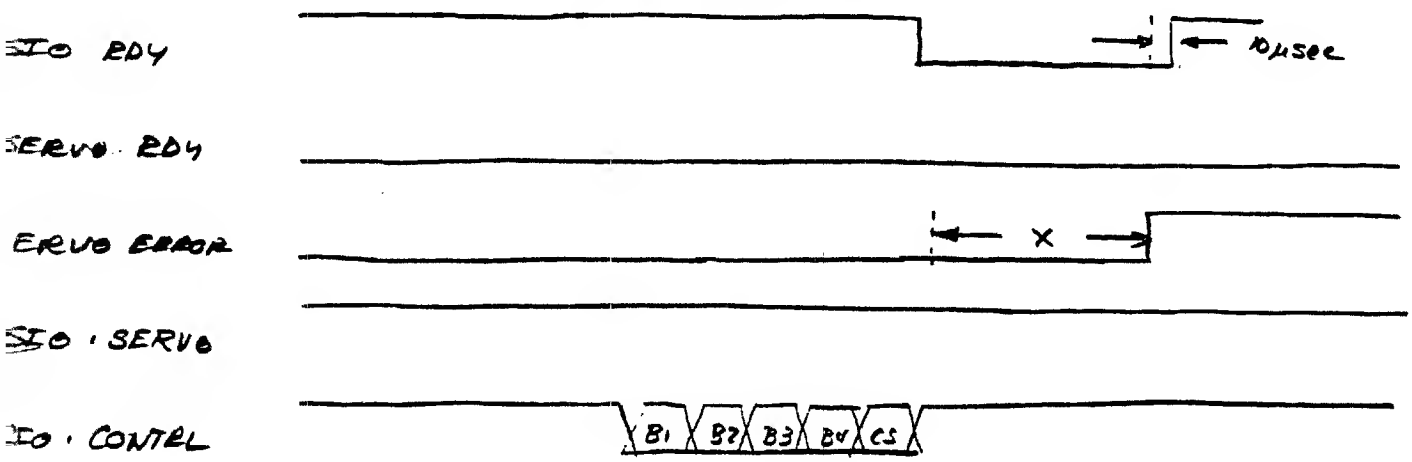
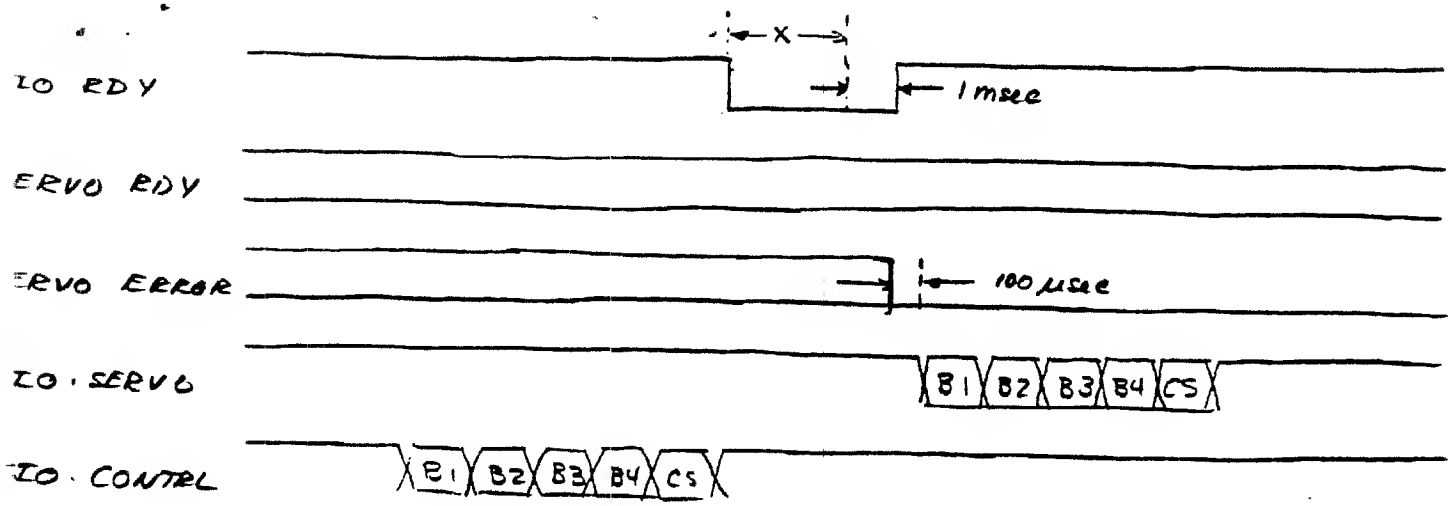
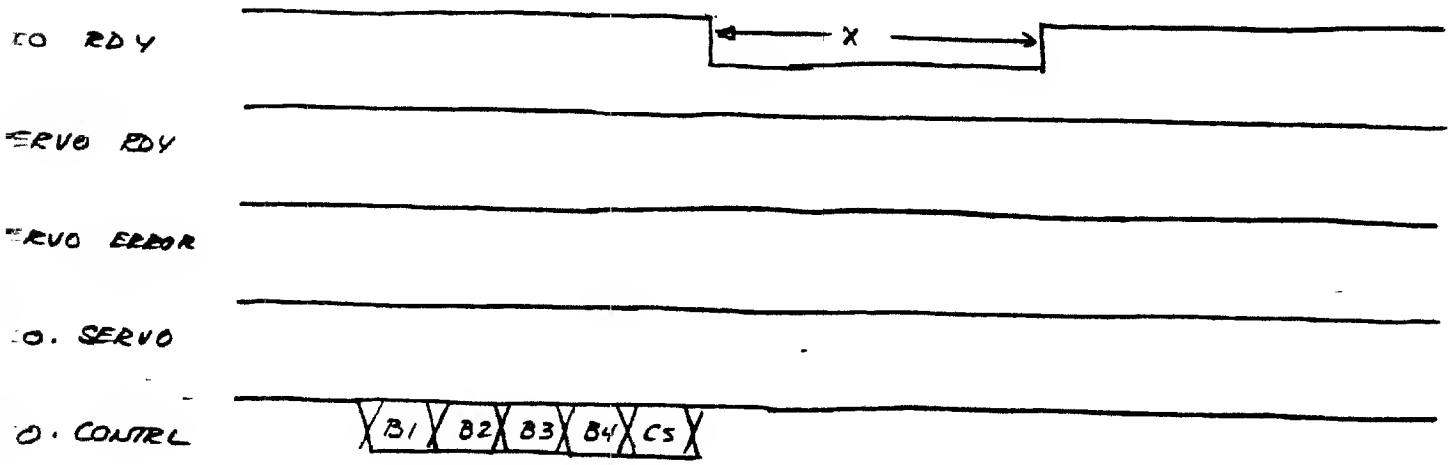


CHART 1

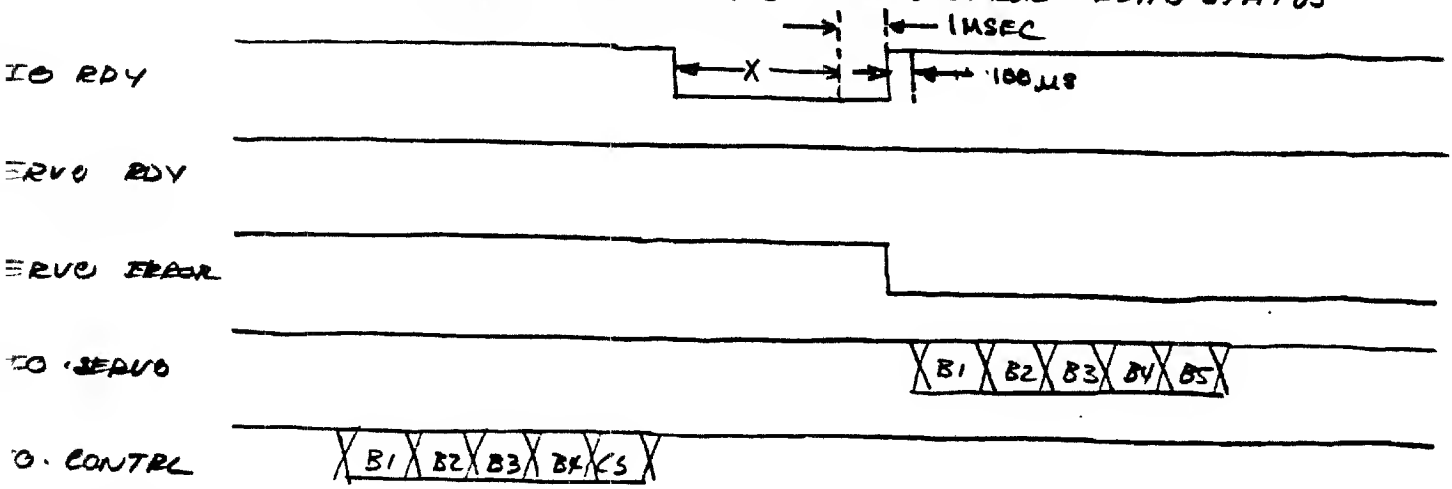
D – READ STATUS COMMAND



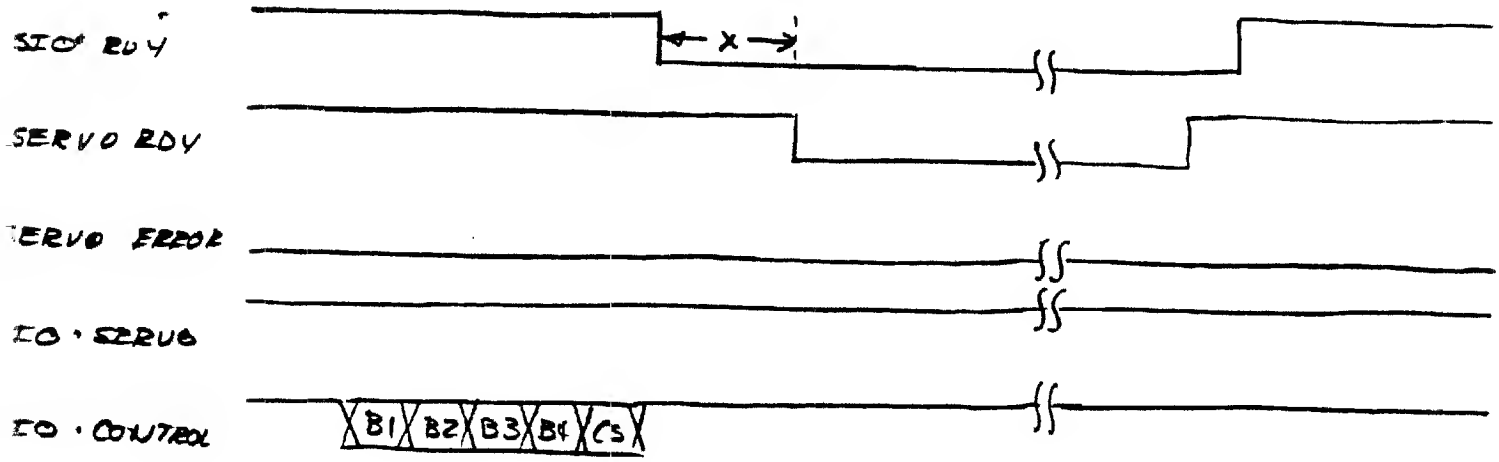
E – TRACK FOLLOWING SERVO ERROR – INVALID COMMAND



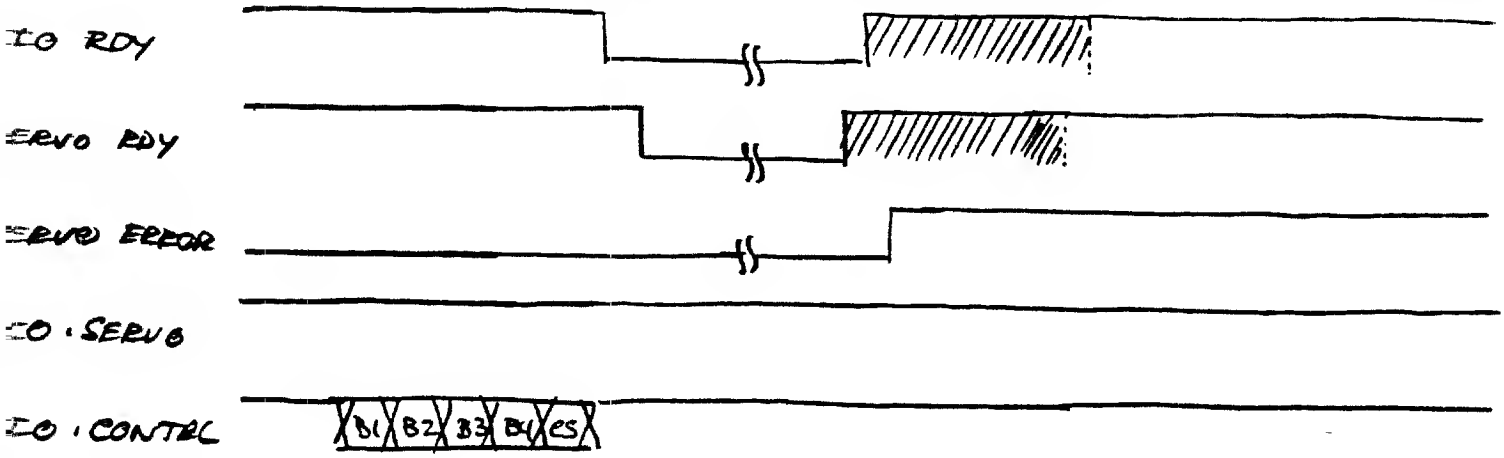
F – TRACK FOLLOWING SERVO ERROR – READ STATUS



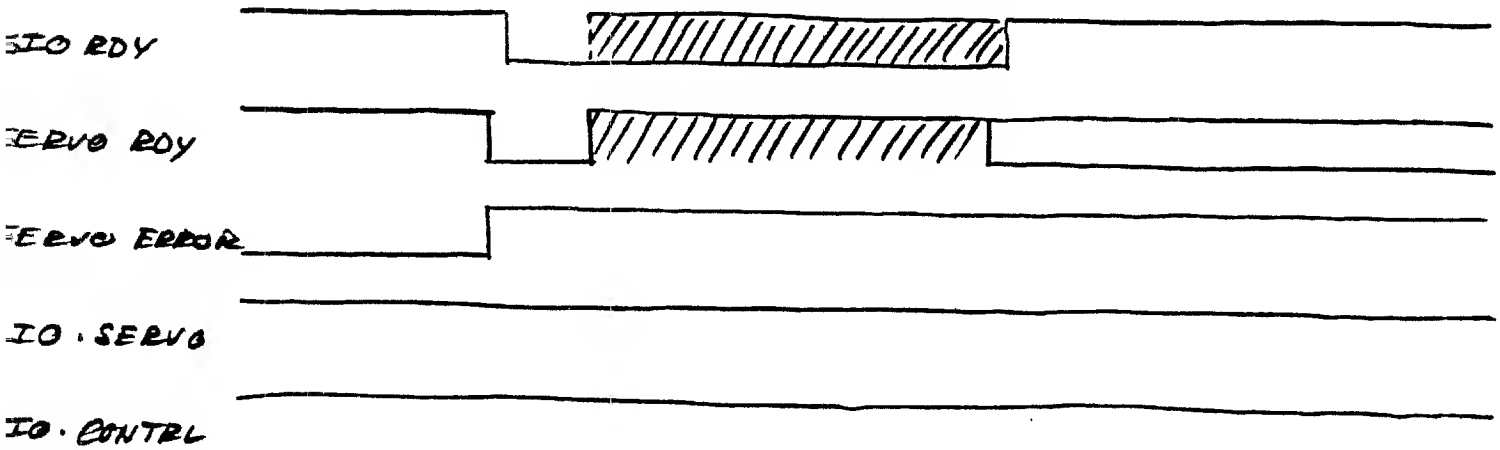
G-TRACK FOLLOWING VALID COMMAND (MOVE)

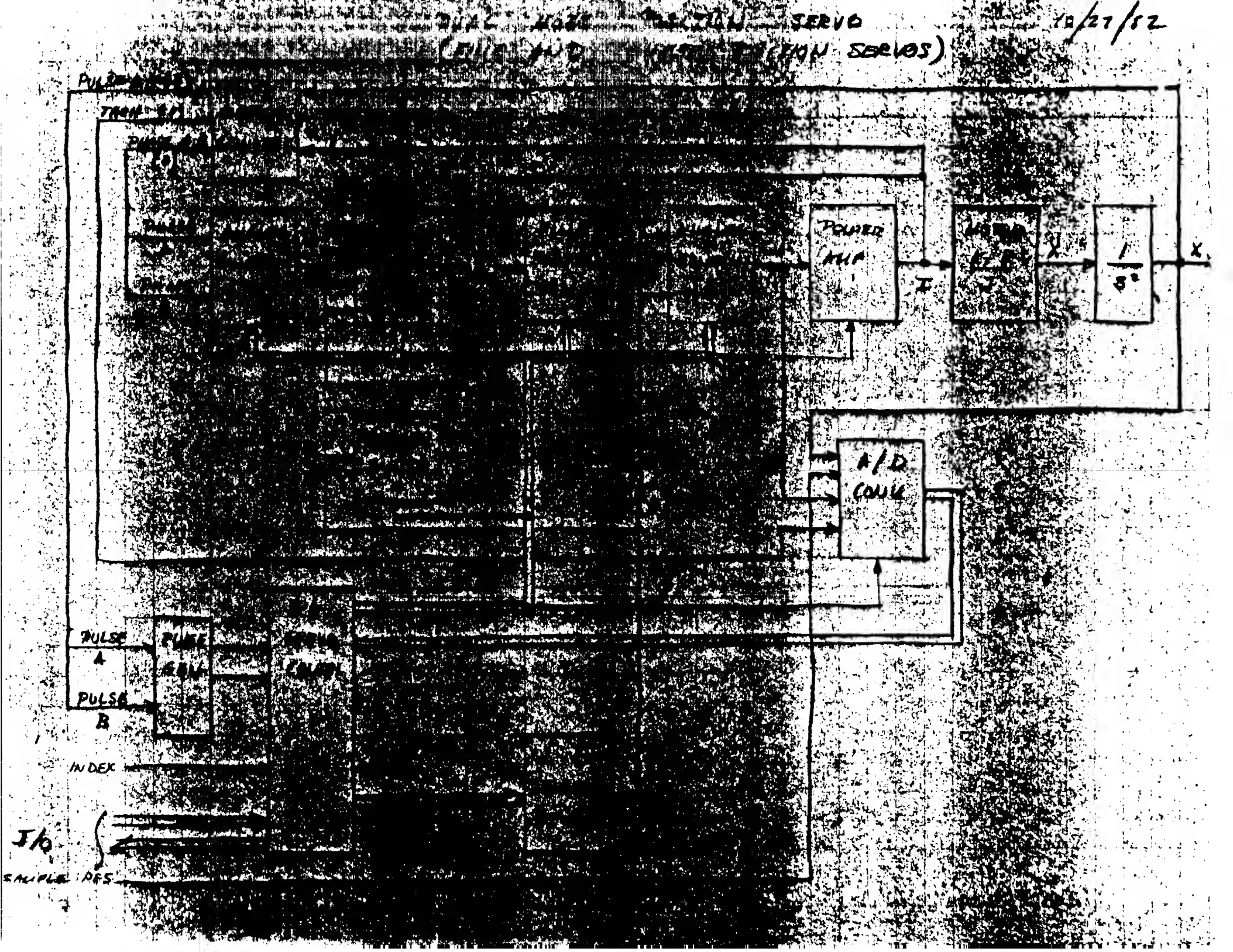


H-TRACK FOLLOWING (MOVE END) FOLLOWED BY SERVO ERROR



I-TRACK FOLLOWING (NO COMMAND) SERVO ERROR




SERVO
10/27/82
(FOUR SERVOS)
POWER AMP
I
X
1/S
X
A/D
COUNTER
PULSE A
PULSE B
INDEX
I/O
SAMPLE RES